LUCAS PROCESSOR ARRAY

Design and Applications

Bertil Svensson

Department of Computer Engineering
University of Lund S-220 07 Lund, Sweden

LUCAS PROCESSOR ARRAY

Design and Applications

Bertil Svensson

To Anna, Jon and Petter

UNIV.
LEIDEN
BIBL.

Printed in Sweden
Studentlitteratur
Lund 1983

CONTENTS

PREFACE 07

Chapter 1 LUCAS PROCESSOR ARRAY 11

1.1 INTRODUCTION 11

1.2 OVERVIEW OF THE LUCAS PROCESSOR ARRAY 17

1.3 DETAILS OF THE DESIGN AND IMPLEMENTATION 24

1.3.1 Processing Elements 24
1.3.2 Memory Modules and input/output structure 32
1.3.3 Interconnection structure 34
1.3.4 Control Unit 48
1.3.5 Host Computer interface 58
1.3.6 Physical Description 60

1.4 COMPARISON WITH OTHER, RELATED DESIGNS 62

1.4.1 STARAN 62
1.4.2 DAP 63
1.4.3 PROPAL 2 64
1.4.4 Vastor 65
1.4.5 CLIP4 66
1.4.6 MPP 66
1.4.7 Conclusion 67

Chapter 2 BASIC INSTRUCTIONS 68

2.1 CLASSIFICATION OF INSTRUCTIONS 68

2.1.1 Basic types of instructions on the associative memory 68
2.1.2 I/O instructions 73

2.2 MOVES, PERMUTATIONS AND MERGES 75

2.2.1 Introduction 75
2.2.2 Basic moves 75
2.2.3 Use of the interconnection network 76
2.2.4 Automatic routing 80

2.3 SEARCHES, COMPARISONS AND LOGICAL INSTRUCTIONS 84

2.3.1 Type <field> --> <selector> 84
2.3.2 Type <field,field> --> <selector> 85
2.3.3 Type <constant,field> --> <selector> 86
2.3.4 A more complex search 87

2.4 ARITHMETIC INSTRUCTIONS 88

2.4.1 Addition and subtraction 88
2.4.2 Multiplication 90

2.5 SUMMARY OF EXECUTION TIMES 95

Chapter 3 SYSTEM SOFTWARE 97

3.1 DEBUGGING MONITOR 97

3.2 ASSEMBLY LANGUAGE FOR MICROPROGRAMMING 98

3.3 HIGH LEVEL LANGUAGE FOR MICROPROGRAMMING 101

3.4 HIGH LEVEL LANGUAGE PASCAL/L 102

Chapter 4 SOME BASIC ALGORITHMS 104

4.1 INTRODUCTION 104

4.2 MATRIX MULTIPLICATION 105

4.2.1 n x n matrices, n processors 106
4.2.2 n x n matrices, n^2 processors 111
4.2.3 n x n matrices, more than n but fewer than n^2 processors 123
4.2.4 n x n matrices, more than n^2 processors 125

4.3 FAST FOURIER TRANSFORM 126

4.3.1 The discrete Fourier transform 126
4.3.2 The fast Fourier transform 127
4.3.3 Implementation on LUCAS 131

4.4 THREE GRAPH-THEORETIC PROBLEMS 137

4.4.1 Shortest path between two given vertices. Unit path length 137
4.4.2 Shortest path between all pairs in a weighted, directed graph 141
4.4.3 Minimal spanning tree 144

4.4.4 Discussion 149

Chapter 5 IMAGE PROCESSING 151

5.1 COMPUTATIONAL DEMANDS IN IMAGE PROCESSING 151

5.1.1 Introduction 151
5.1.2 General demands on a computer for image processing 153

5.2 DIFFERENT ATTEMPTS TO MEET THE DEMANDS 154

5.2.1 Fast neighbourhood access 154
5.2.2 A small number of special purpose processors 155
5.2.3 A large number of conventional microprocessors 156
5.2.4 A very large array of simple processors 157
5.2.5 LUCAS compared to other machines 158
5.2.6 The advantages of image parallelism 158

5.3 ORGANIZATION OF PROCESSOR ARRAYS FOR IMAGE PROCESSING 160

5.3.1 Introduction 160
5.3.2 Two-dimensionally organized arrays 161
5.3.3 Linearly organized arrays 162
5.3.4 Discussion 165

5.4 IMAGE OPERATIONS ON LUCAS ORGANIZED AS A LINEAR ARRAY OF PROCESSING ELEMENTS 168

5.4.1 Introduction 168
5.4.2 Genuinely local operations. Small neighbourhood sizes 171
5.4.3 Genuinely local operations. Larger neighbourhood sizes 189
5.4.4 Semi-local operations 192
5.4.5 Measurements 204
5.4.6 Global transforms 209
5.4.7 Input/output 211
5.4.8 Larger images 212
5.4.9 Comparison of execution times 215

5.5 CONCLUSIONS 219

Chapter 6 CONCLUSIONS AND FUTURE RESEARCH 220

6.1 DESIGN 220

6.1.1 Processing Elements 220

6.1.2 Communication 220
6.1.3 Control Unit 222

6.2 IMPLEMENTATION 224

6.2.1 Experience 224
6.2.2 Possibilities offered by VLSI technology 225

6.3 APPLICATIONS 232

6.4 PROGRAMMING 233

REFERENCES 234

APPENDIX 239

A1. SCHEMATIC DRAWINGS OF A PROCESSOR BOARD 239

A2. EXAMPLES OF MICROPROGRAMS 245

PREFACE

The subject of this thesis can be introduced without referring to computers. Let us imagine five hundred people gathered in an open place, say at Speaker's Corner of Hyde Park. Furthermore, let us imagine a speaker that all the people can hear and understand. How can the speaker get to know who is the oldest in the audience? How can he find out how many have a yellow pencil in their pocket? Or, suppose he wants one person of each of the nationalities represented in the crowd to stand up, and everyone else to sit down. What instruction does he give to the crowd?

Evidently, this depends strongly on what the speaker thinks of his audience. Most probably, he does not presuppose any intellectual capacity at all (except that people know their nationality, how old they are and if they have a yellow pencil in their pocket). The speaker, being the only one capable of counting things and comparing ages and nationalities, would then walk around in the crowd and ask questions and give orders until his goal was achieved.

Instead, if he thinks better of his audience, could he then solve this kind of problems more efficiently? The oldest-person problem, for example, is solved efficiently by the following procedure: First, those under 50 years of age are told to sit down. If there are still people that stand up, those under 75 should sit down. If nobody is now standing, those over 62 should get up, and so on, until finally only one is standing. This solution requires no passing of information between the listeners, but the most efficient solution to the yellow-pencil problem does. Therefore, to solve that problem the crowd must be organized in some way. We may then ask: Can this be done in a way that is useful for many different tasks?

Questions concerning ways of organizing a large amount of data

processing units, and ways of instructing the units what to do, are becoming increasingly important in computer science research. This thesis addresses such questions under certain restrictions. It is assumed that all processing units are identical and that they all get the same instructions from a central control unit. The research has involved the implementation of a computer, called LUCAS, that works according to these principles. The thesis is organized as follows:

Chapter 1 describes the LUCAS system and motivates the solutions used in the design. The emphasis is on the facilities of the individual processors and the network that allows them to exchange data. In Chapter 2 the basic instructions of the processor array are described.

Chapter 3 is an overview of the software developed for LUCAS and introduces languages that are used in the thesis for the description of algorithms. My contribution to the development of the software is minimal and the material is included only to make examples understandable.

Chapters 4 and 5 discuss the usefulness of this type of processor in different computational areas. Matrix multiplication and Fourier transformation are frequently used mathematical tools. Graph theoretic problems are encountered in many situations. Image processing deals with enormous amounts of data and parallel processing is required. We show that the computer structure that we have implemented meets the demands from the different areas well.

In Chapter 6 we comment on the results and on the design. We also examine the potential of VLSI technology for implementing very large arrays of the same type as LUCAS.

The writing of the thesis and the performance of the research would not have been possible without the help from many people. It is a pleasure to thank Dr. Rolf Johannesson for guidance and encouraging support during the course of the work and for constructive criticism of the manuscript. I am deeply indebted to my colleagues Ivan Kruzela and Christer Fernström with whom I have

performed the work. Without the specific qualities of Ivan and Christer nothing of this would have been possible. Ivan was the one who suggested the topic and Christer was the one who never found the obstacles that we encountered too difficult to surmount. Thank you, Ivan and Christer.

I also want to thank Lennart Ohlsson for the many valuable discussions on the contents of the thesis that we have had during the writing. Many ideas originate from Lennart.

Thank you, Lena Månsson for the high quality drawings that you have made, improved not only in form, but also in content, compared to my sketches.

Anders Ardö has provided the editing tools necessary for producing all this text and has always been willing to make modifications according to my wishes. Thank you, Anders.

Dan Leek has mounted most of the circuits on the boards of LUCAS and assisted with many practical details. Per Westerberg and Josef Wajnblom have done careful wrapping of thousands of wires. Thank you.

I would also like to thank Mats Cedervall for assistance with the use of the department's computer system, Kerstin Johnsson for typing parts of the manuscript and Lars Nielsen for valuable suggestions on the image processing part.

Many more at the Department of Computer Engineering and the Center for Industrial Computer Systems have helped with good advise and support and all this is very much appreciated.

I would also like to thank my family for patient endurance when the machine would never work and the writing would never end.

The financial Support given by the Swedish National Board for Technical Development through the Center for Industrial Computer Systems is greatly appreciated.

CHAPTER 1
LUCAS PROCESSOR ARRAY

1.1 INTRODUCTION

The requirements on the performance of computers are steadily growing. New application areas are considered that pose performance requirements earlier thought of as unrealistic. In the history of computing, growing demands have to a substantial degree been met through increased circuit speed. In the most powerful computers of each time, however, parallelism has also been introduced because improvements in circuit speed alone have not been sufficient to produce the required performance.

History shows that concepts introduced in the high-performance computers often a few years later become part of the design of more moderately sized - or at least more moderately priced - wide-spread computers. The rapid progress of the Very Large Scale Integration (VLSI) technology also paves the way for the use of parallelism.

New computer architectures often originate from needs to solve problems from some specific application areas efficiently. This makes them in a way tuned to those specific problem classes. However, many architectures are of a general purpose kind or show great similarities with each other. Therefore, a need to find efficient algorithms to solve problems from different areas, on specific classes of parallel machines, is clearly seen. Existing programming languages are strongly influenced by classical computer architecture and are not suited for expressing these algorithms. Thus, there is also a need for new languages.

Parallelism can be introduced in a computer design in different ways. Conventional computers use a form of parallelism that can be called 'bit parallelism on the level of a data item', i.e. the different bits of a data item are processed in parallel by the arithmetic-logic unit. Common degrees of this parallelism range from 8 to 64 bits. On problems with low precision data the parallelism is often poorly utilized. Analysis of such problems that generally require extremely long computation times very often shows that there is another form of parallelism that can be utilized, which is orders of magnitude larger. It is the possibility to treat many data items simultaneously, independent of each other. We call this 'data parallelism on the instruction level'. It is frequent in computational areas like image and signal processing, data base management, and those involving matrix manipulation and solution of differential equations. The degree of parallelism offered in these applications often is in the range between a thousand and several million. Thus, a computer with some thousand processing elements is very likely to be fully utilized in many applications. For dedicated designs an even larger degree of parallelism can be motivated.

Computer architectures of this kind fall into the category called SIMD (Single Instruction stream - Multiple Data stream) in the classification scheme of Flynn (1966). The basis of this scheme is that a processor of any kind processes data by a sequence of instructions. Based on the context of a data stream and an instruction stream, four possibilities exist:

Single Instruction stream - Single Data stream (SISD)
Single Instruction stream - Multiple Data stream (SIMD)
Multiple Instruction stream - Single Data stream (MISD)
Multiple Instruction stream - Multiple Data stream (MIMD)

The four categories are illustrated in Figure 1.1.

The ordinary von Neumann architecture belongs to the SISD category. In a SIMD architecture each processing unit executes the same instruction, but on different data. In MIMD systems many

processors cooperate to solve a common computational task, but the tasks assigned to the individual processors can all be different. The exact structure of the MISD architecture is not fully agreed upon. Most authors put the pipelined processing structure shown in the figure in this category. Others claim that pipeline processors should be considered as examples of SIMD machines (e.g. Baer, 1980). In that case the MISD category becomes empty, at least if only existing machines are considered.

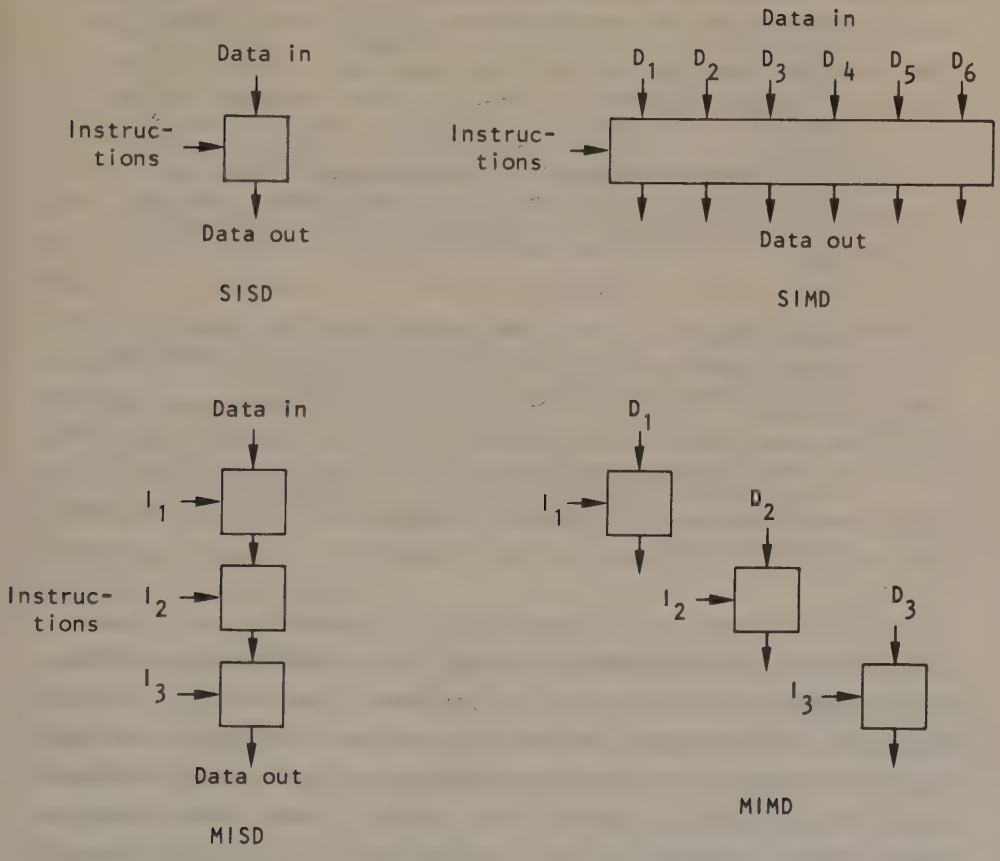


Figure 1.1 Flynn's classification scheme

The necessity of abandoning the von Neumann architecture when designing high-performance systems has been advocated by many authors. One of the most prominent, John Backus, maintains that we are also hampered in our way of designing algorithms by the habit of always breaking them down into sequential form. We cite a few lines from his Turing Award Lecture (Backus, 1978):

> In it simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the von Neumann bottleneck. The task of the program is to change the contents of the store in some major way: when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.
> ....
>
> Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is the tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. (p 615)

This view points to the importance of implementing new computer architectures and using them in practice. Many computational problems have engaged a large number of computer scientists for decades because of the relevance of the problems for computations on von Neumann computers. With new architectures they may be less important while others become essential. For example, when working with a highly parallel computer, we may find sorting to be of little interest while the problem of routing large amounts of data between different parts of the machine without conflict becomes a salient problem.

In this thesis we present design principles used in, and conclusions drawn from, a project that includes the implementation of

a general purpose parallel computer, LUCAS (Lund University Content Addressable System). The number of processing elements in the design is not limited in itself but has been fixed to 128 in the implemented version used for application studies. Algorithms mainly within three large areas - image processing, signal processing, and data base management - have been programmed. The results summarized in this thesis mainly concern the image processing area and to some extent signal processing and other areas. Implementations of operations to manipulate relational data bases are described in (Kruzela, 1983) and to some extent in Kruzela and Svensson (1981). A high level language for programming of LUCAS is presented in (Fernstrom, 1983) as well as further details of the implementation and system software.

The main object with the design and implementation of LUCAS is to make available a research vehicle for the study of architectural principles, programming methodology and applicability of a certain kind of parallel computers. With certain principles and design details fixed (like bit-serial working mode, the use of conventional memory circuits, SIMD organization) we want to be able to modify parts of the architecture, to see how they best suit certain applications. These parts include the network that interconnects the processing units, the input/output system, and the instruction sets at different architectural levels. Furthermore, the effects on execution time and utilization efficiency of different ways of communication with the host computer can be studied.

The size of LUCAS was decided from application requirements. The machine must be large enough to give a substantial performance improvement over conventional medium-size computers in important application areas. The implementation shows that this is possible at a reasonably low cost.

Computers using the same principles as LUCAS (bit-serial SIMD) have been built earlier. They have all been supercomputers, like STARAN (Batcher,1974,1979) and DAP (Flanders et al., 1977), or dedicated computers, like CLIP4 (Duff,1979) and the recent design MPP (Batcher, 1982).

The first thoughts on the design of LUCAS where raised in 1978. The initial ideas had the signal processing area in mind, especially the FFT. Computer simulations were made in order to find a suitable instruction set. A small prototype comprising eight processing elements was built in 1979. Experience from this prototype had great impact on the final design which was settled in 1980. Gradually, the machine became functioning with more and more processing elements; in the spring 1982 it was running reliably with all its 128 processing elements. Software development took place in parallel with hardware implementation during this period, and this means software both for hardware debugging and algorithm development.

Three persons have been working on the project during the whole time.

1.2 OVERVIEW OF THE LUCAS PROCESSOR ARRAY

The LUCAS system consists of three parts: the Host Computer, the Control Unit and the Processor Array (Figure 1.2.1). In addition to this a special purpose I/O processor may be added in order to provide fast input/output.

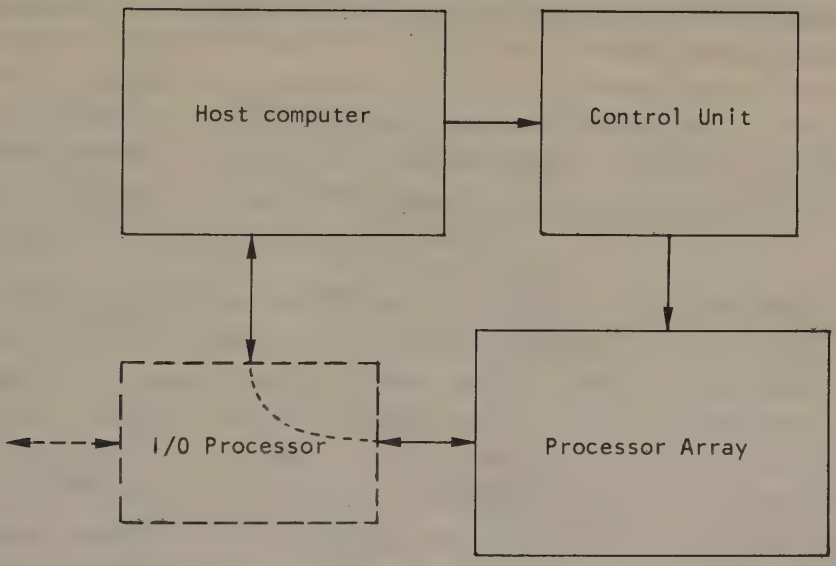


Figure 1.2.1 Overview of the LUCAS system.

The Processor Array (Figure 1.2.2) consists of 128 Memory Modules, each with a Processing Element (PE). The design does not put a limit on this number. Thus, performance may be increased through the addition of more of these modules. A Memory Module is a 4096 bit memory, where one bit is accessible at a time. All memory modules receive the same address from the Control Unit, i.e. a bit-slice of the whole memory area is accessible at a time as indicated in Figure 1.2.2. The Processing Elements work on one-bit data and have four internal one-bit registers. Most often a PE uses data from "its own" memory module. However, an interconnection network is included that provides possibilities for data access from other modules as well. The system works in full synchronism and the processing elements all get the same control signals each clock cycle from the control unit.

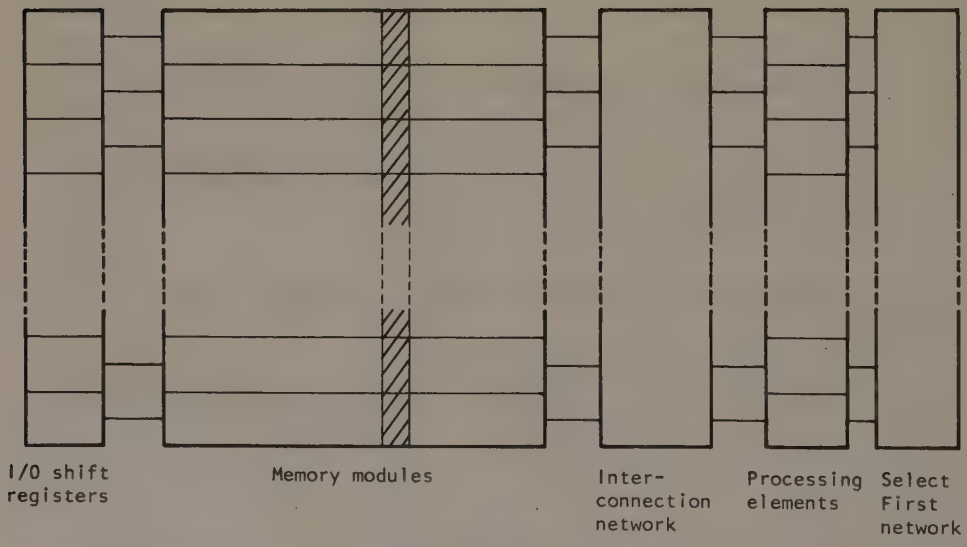


Figure 1.2.2 Schematic drawing of the Processor Array.

The ensemble of memory modules may be seen as one single content addressable memory (associative memory). A typical case when

this view is adequate is when a command is issued to all processing elements to compare the memory contents with a certain template, broadcast to all PEs. Match/non-match is marked in one of the one-bit registers, the tag flip-flop, in each PE. Often, data from the memory modules that meet the search criteria are to be read out to the Host. If this is to be done sequentially, a "multiple match resolver" has to be included. This takes the form of a Select First-network, a chain of gates from PE to PE, which selects the first one of those processing elements that have their tag flip-flops set to one. It can be used iteratively to select the PEs in sequence.

In some applications, e.g. data base management, the way of looking at the processor array as an associative memory is very natural. In many other applications, however, viewing the array as an ordered ensemble of processors attached to memory modules is more appropriate. The associative memory approach alludes more to the way the information is retrieved from the array, while the processor array approach more is a description of the machine organization. We will mainly use the latter in this thesis.

With every memory module there is an 8-bit I/O shift register. The I/O shift registers have two purposes. Firstly, they are used for input to and output from the memory modules. They are accessed like ordinary memory cells from the Host computer (or I/O processor) and communicate with the memory modules in a word-parallel, bit-serial fashion. Secondly, they provide an additional means of communication between memory modules. One byte of any module can be transferred to all the other modules or to a selected subset of modules according to the state of the tag flip-flop in each PE.

In the implementation of the processor array some details are fixed while others are modifiable. The main working mode (bit-serial, common memory address, common control signals) is fixed. The interconnection network can easily be reconfigured but it is not software programmable. The functions implemented in the processing elements are defined by programmable read-only memories. Thus, they may also be altered in order to suit the demands of different

applications, or simply as a result of growing experience.

The purpose of the Control Unit (Figure 1.2.3) is to receive instructions from the Host computer and to interpret them in terms of microinstructions. The microinstructions partly direct the sequencing of the Control Unit itself, partly direct the work of the Processor Array. The set of control signals sent to the Processor Array consists of (1) the bit address to the Memory Modules, (2) a function code for the Processing Elements, (3) an interconnection selector code and (4) control of memory writing, the I/O shift registers and the Select First-network. The execution of an instruction is controlled by the contents of a microprogram memory, the size of which is 4096 words, 80 bits wide. The microprogram memory is of read/write type and is loaded from the Host.

The Instruction Register, IR, holds the current instruction. The locations and length of the operands are specified in the Parameter Registers, PR1..PR4. As an example, parallel addition of two vectors is shown in Figure 1.2.4. An important feature of LUCAS, illustrated in this example, is the ability to treat operands of different lengths with the same instructions. No extra bits that carry no information have to be brought through the computations.

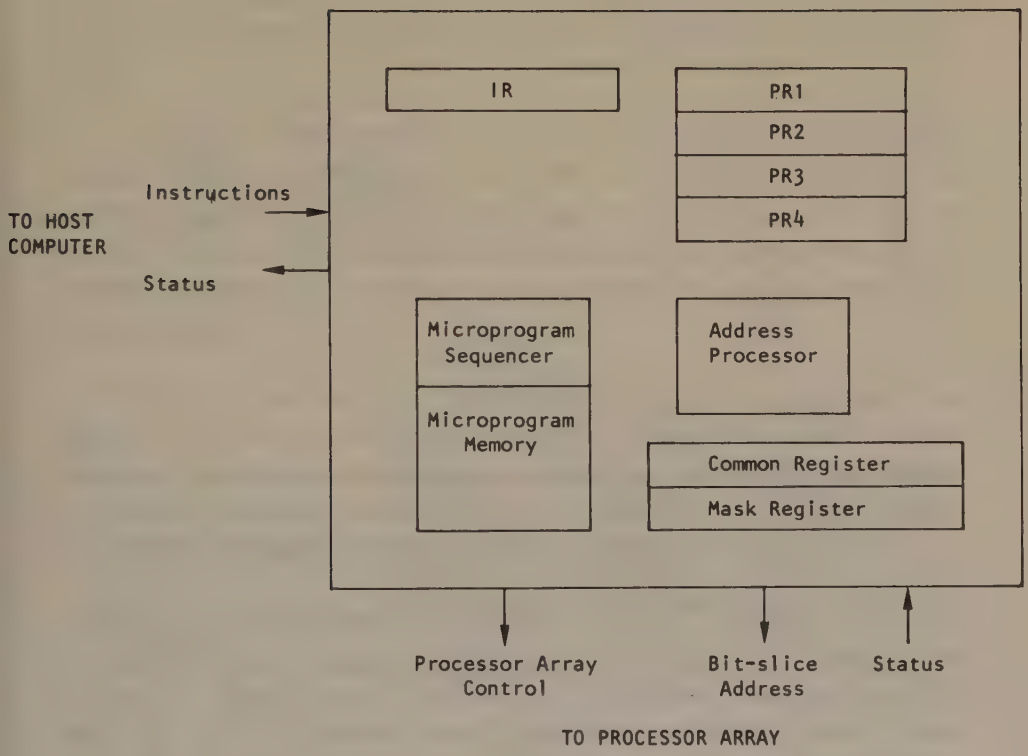


Figure 1.2.3 The Control Unit

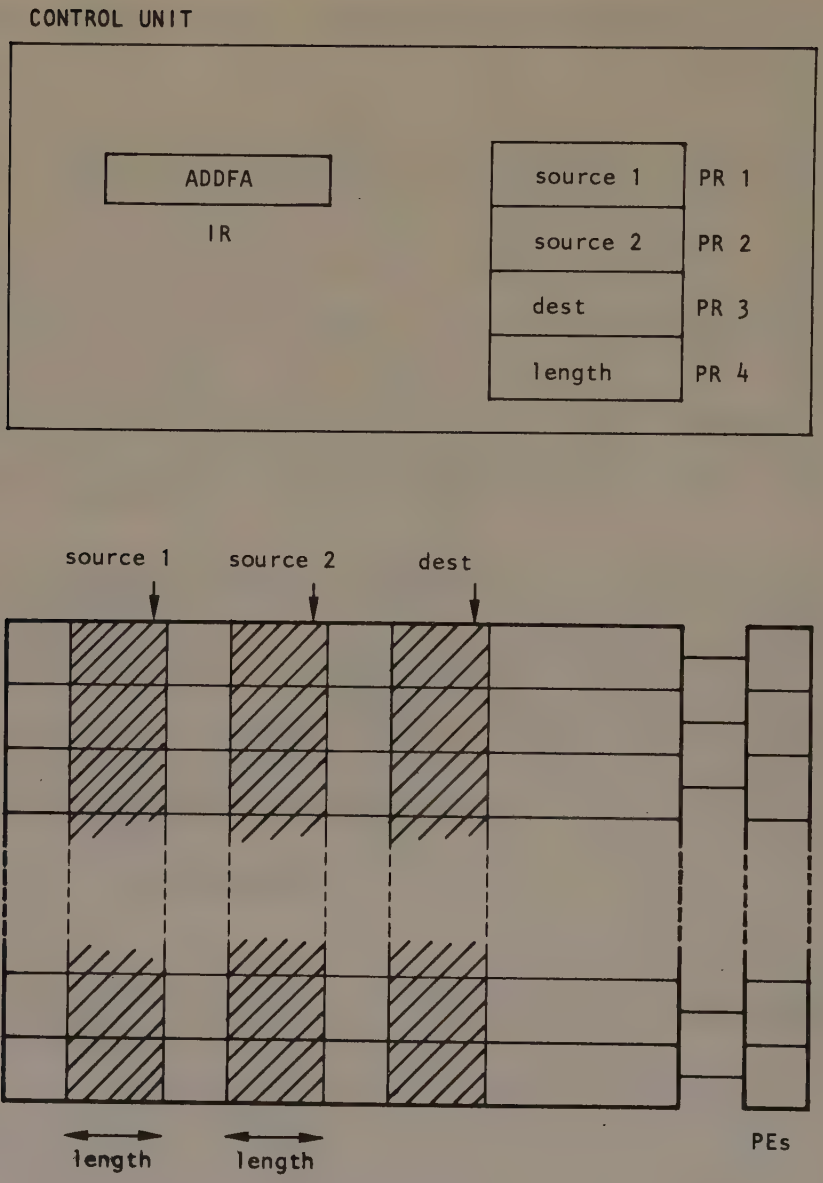


Figure 1.2.4 Illustration of the four-parameter instruction 'add fields all': ADDFA source1, source2, dest, length.

An important part of the Control Unit is the Address Processor, AP. Its task is to make fast computations of addresses to bit-slices in the Memory Modules (increment, decrement, add constant, compare, etc.). The Address Processor, which operates on 12 bit long words contains 16 general purpose registers and a 32 words external stack memory.

Furthermore, the Control Unit contains a Common (Comparand) Register and a Mask Register, each 4096 bits wide, thus matching the length of the Memory Modules. The Common Register is used to hold an argument that is to be used in the computations of all the PEs, e.g. a search argument or a constant to be added to a data item in each memory module.

Through a test input to the Sequencer the contents of the Mask Register influences the flow of microinstructions. The normal use of the Mask Register is to mask out certain bits in search operations. The Common and Mask Registers are connected to I/O shift registers in exactly the same way as the Memory Modules.

The Instruction Register and the Parameter Registers each have an address in the memory address space of the Host, and are loaded by ordinary memory write instructions. This is the case also with the 128+2 I/O shift registers. Thus, these can be loaded directly from secondary memory, using DMA, without passing through the Host.

In the present implementation an ordinary microcomputer system serves as the Host computer.

1.3 DETAILS OF THE DESIGN AND IMPLEMENTATION

1.3.1 Processing Elements

A Processing Element (Figure 1.3.1) consists of four parts:

* A set of one-bit registers: T(tag), R(result), C(carry), and X(auXiliary).
* An Arithmetic Logic Unit (ALU).
* A Data Selector.
* Part of a network that implements the SELECT FIRST and SOME functions.

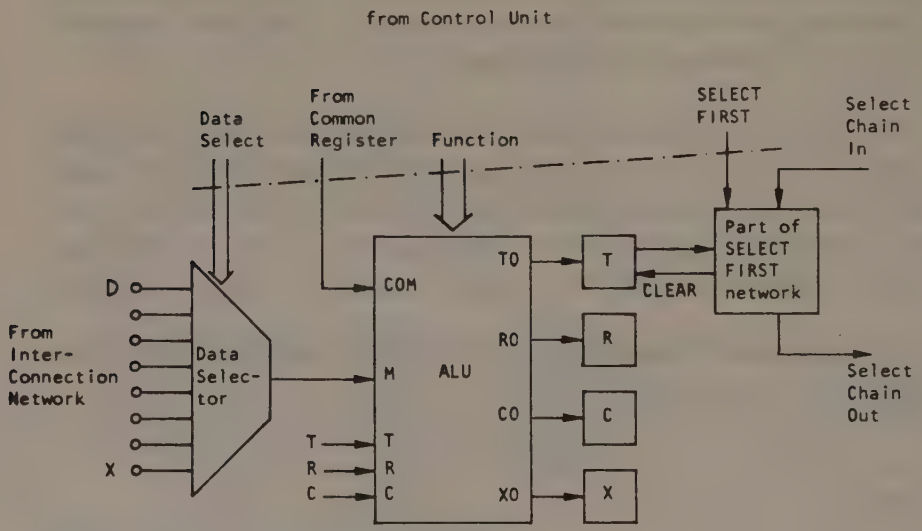


Figure 1.3.1 A Processing Element

Registers

The four registers, T, R, C, and X, have slightly different features.

The T register, called the Tag flip-flop, has its output connected to the write control logic of the corresponding memory module. Thus it can be used as activation control, inhibiting change in the memory of those PEs where T equals zero. Furthermore, the Tag flip-flop is connected to the SELECT FIRST-network, which on a control signal from the Control Unit resets all Tag flip-flops but one, namely the one with the lowest number in the linear ordering of the PEs. The Tag flip-flop is also the input to the SOME- network. This network indicates to the Control Unit whether some or none of the Tag flip-flops are set. Finally, the Tag flip-flop controls the I/O register of the Memory Module in a way described below.

The R (Result) register is the only register from which data may be written directly into the memory. Its output may be put in a high-impedance state by a control signal in order to permit writing in the memory from another source - the I/O register.

The C (Carry) register is a general purpose register with no special features. In arithmetic operations it is used to hold the carry bit.

The X (AuXiliary) register is also general purpose. Its output is not directly connected to the ALU. Instead, one of the Data Selector inputs must be used, as shown in Figure 1.3.1.

Arithmetic Logic Unit

A bipolar Programmable Read Only Memory (PROM) serves as arithmetic logic unit. The size of the PROM is 1024 words of 4 bits each. Five inputs are used to specify the function; thus 32 functions are available. All PEs receive the same function code. The remaining

five inputs are data inputs.

Since the functions are specified by the contents of a PROM they can easily be altered to suit a specific application. An "ALU assembler", implemented on a Nova minicomputer, generates the PROM contents when given as input boolean expressions for the four outputs.

One general purpose function set has been used for most of the time that the machine has been operable. It has proved to be applicable in a wide variety of computational areas, including image processing, signal processing and data base management. The function set is listed in table 1.3.1. Some functions are listed more than once, and with different mnemonics, since they naturally belong to more than one group in the function set. In table 1.3.1, registers not mentioned are left intact. X0=M leaves the X-register intact if, by the Data Select code, M is chosen to be X (which is the default value from the microprogram assembler).

The Data Selector

The Data Selector allows a Processing Element to receive data from eight different memory modules. Which memory modules that are available depends on the wiring of the interconnection network. One of the eight sources is fixed to be the output of the associated memory module (the D input). The remaining seven inputs can be wired to any source. All PEs receive the same Data Select code.

The SOME and SELECT FIRST networks

The output of the SOME-network tells the Control Unit whether some or none of the Tag flip-flops are set. It has the structure shown in Figure 1.3.2, giving a depth of 16 gates when the number of PEs is 128.

When the Control Unit issues the SELECT FIRST command, only

one (the first) Tag flip-flop is to remain set to one, all others are reset. This is done by an iterative combinatorial network with the states of all the Tag flip-flops as inputs and CLEAR signals to the flip-flops as outputs. To reduce the depth of the network, the outputs of the 8-input gates in the SOME-network are used as a kind of look-ahead. The structure of the SELECT FIRST network is shown in Figure 1.3.3.

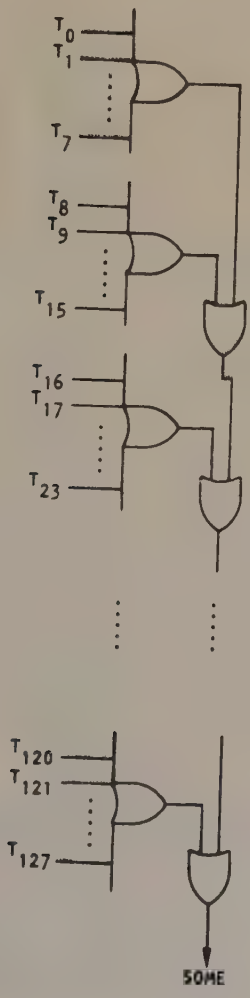


Figure 1.3.2 The SOME-network

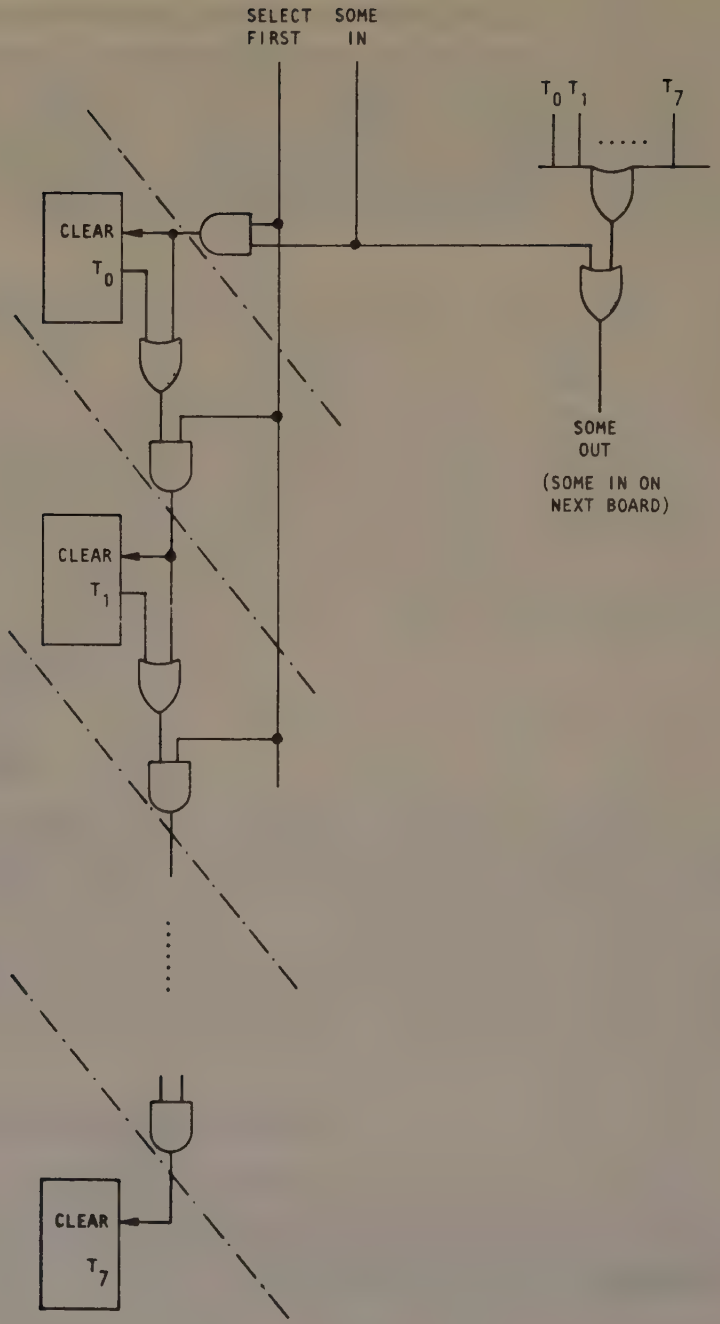


Figure 1.3.3 The SELECT FIRST- and SOME- network of a board of eight consecutive Processing Elements.

GROUP	MNEMONICS	EXPLANATION	EXPRESSIONS		NOTE
No operation	NOP	No operation	XO=M		=LXMA
Set, Clear, Complement	SETT	Set tags	TO=1	XO=T	
	COT	Complement tags	TO=T'	XO=M	
	SCA	Set C, all	CO=1	XO=M	
	CCA	Clear C, all	CO=0	XO=M	
	CRA	Clear R, all	RO=0	XO=M	
	CORA	Complement R, all	RO=R'	XO=M	
	CORT	Complement R, tagmasked	Where T=1 RO=R', elsewhere RO=R	XO=M	=XORRTA
Load flip-flops	LRMA	Load R from M, all	RO=M	XO=R	
	LRMT	Load R from M, tagmasked	Where T=1 RO=M, elsewhere RO=R		
	LRCA	Load R from C, all	RO=C	XO=M	
	LRTA	Load R from T, all	RO=T	XO=M	
	LTMA	Load T from M, all	TO=M	XO=T	
	LTMT	Load T from M, tagmasked	Where T=1 TO=M, elsewhere TO=T=0	XO=T	=CMOT =ANDTMA
	LTRA	Load T from R, all	TO=R	XO=M	
	LTRT	Load T from R, tagmasked	Where T=1 TO=R, elsewhere TO=T=0	XO=M	=CROT =ANDTRA
	LTMIT	Load T from M inverted, tagmasked	Where T=1 TO=M', elsewhere TO=T=0	XO=T	=CMZT =ANDTMIA
	LTRIT	Load T from R inverted, tagmasked	Where T=1 TO=R', elsewhere TO=T=0	XO=M	=CRZT =ANDTRIA
	LCRA	Load C from R, all	CO=R	XO=M	
	LXMA	Load X from M, all	XO=M		=NOP
	XRT	Exchange R and T	TO=R	RO=T XO=M	

TABLE 1.3.1 Part 1

Compare (Result in T)	CRZT	Compare R to Zero, tagmasked	Where (T=1 and R=0) TO=1, elsewhere TO=0 XO=M		=LTRIT =ANDTRIA
	CROT	Compare R to One, tagmasked	Where (T=1 and R=1) TO=1, elsewhere TO=0 XO=M		=LTRT =ANDTRA
	CRMT	Compare R to M, tagmasked	Where (T=1 and R=M) TO=1, elsewhere TO=0 XO=T		
	CRCT	Compare R to COM, tagmasked	Where (T=1 and R=COM) TO=1, elsewhere TO=0 XO=M		
	CMZT	Compare M to Zero, tagmasked	Where (T=1 and M=0) TO=1, elsewhere TO=0 XO=T		=LTMIT =ANDTMIA
	CMOT	Compare M to One, tagmasked	Where (T=1 and M=1) TO=1, elsewhere TO=0 XO=T		=LTMT =ANDTMA
	CMCT	Compare M to COM, tagmasked	Where (T=1 and M=COM) TO=1, elsewhere TO=0 XO=T		
Logical	ANDTRA	AND T with R, all	TO = T AND R	XO=M	=LTRT =CROT
	ANDTMA	AND T with M, all	TO = T AND M	XO=T	=LTMT =CMOT
	ANDTMIA	AND T with M inverted, all	TO = T AND M'	XO=T	=LTMIT =CMZT
	ANDTRIA	AND T with R inverted, all	TO = T AND R'	XO=M	=LTRIT =CRZT
	ANDRMA	AND R with M, all	RO = R AND M	XO=R	
	ORRMA	OR R with M, all	RO = R OR M	XO=R	
	XORRMA	XOR R with M, all	RO = R XOR M	XO=R	
	XORRTA	XOR R with T, all	RO = R XOR T	XO=M	

TABLE 1.3.1 Part 2

Arithmetic	ADMA	Add M to R with carry, all	RO=M+R+C CO=MRvMCvRC XO=Overflow = MR(RO)'vM'R'(RO)	+ is mod 2 add
	ADMIA	Add M inverted to R with carry, all	RO=M'+R+C CO=M'RvM'CvRC XO=Overflow = M'R(RO)'vMR'(RO)	" "
	ASMT	Add/sub M to/from R with carry where T=1/0	Where T=1: same as ADMA Where T=0: RO=M+R+C CO=R'MvC(R+M)' XO=Overflow = R'M(RO)vRM'(RO)'	" "
	ACMA	Add COM to M with carry, all	RO=COM+M+C CO=(COM)Mv(COM)CvMC XO=Overflow = M(COM)(RO)'vM'(COM)'(RO)	" "
	ACIMA	Add COM inverted to M with carry, all	RO=(COM)'+M+C CO=(COM)'Mv(COM)'CvMC XO=Overflow = M(COM)'(RO)'vM'(COM)(RO)	" "
	ACMIA	Add COM to M inverted with carry, all	RO=COM+M'+C CO=(COM)M'v(COM)CvM'C XO=Overflow = M'(COM)(RO)'vM(COM)'(RO)	" "

TABLE 1.3.1 Part 3

1.3.2 Memory Modules and input/output structure

With each of the Processing Elements there is a Memory Module with 4096 bits of storage and a facility for input and output of data. Figure 1.3.4 shows one of the modules.

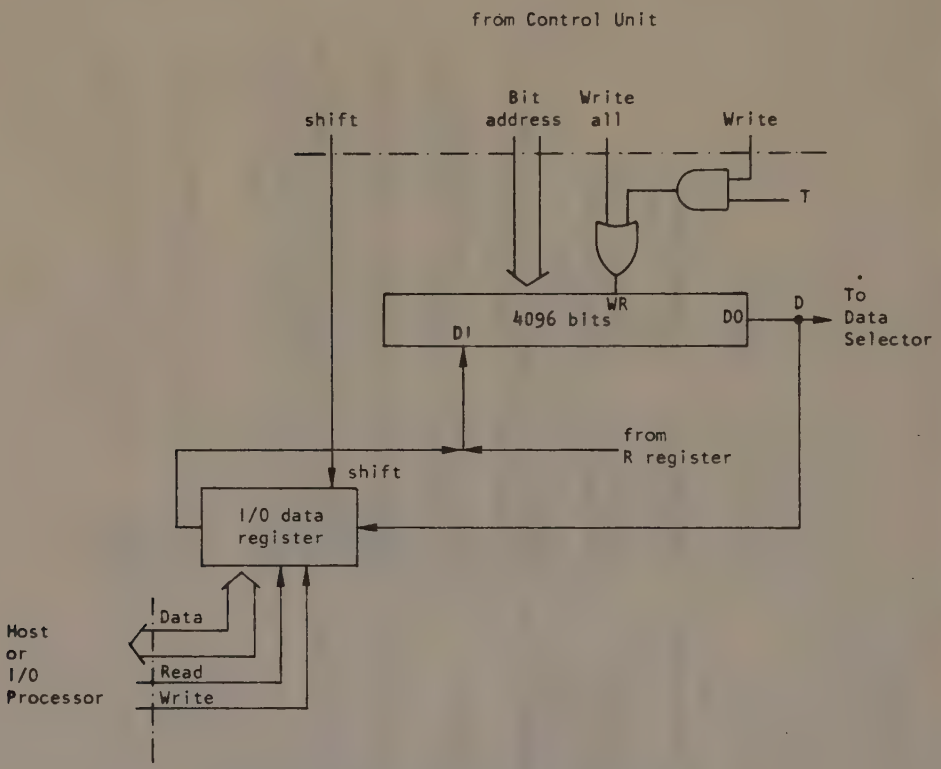


Figure 1.3.4 A Memory Module with its I/O data register.

The memory, which is an ordinary 4096 x 1 read/write memory chip with 55 ns access time, receives a 12 bit address from the Control Unit and two different write signals. When "write all" is activated all memory modules are written into, whereas "write" gives writing in a module only if the Tag flip-flop of the Processing Element is set to

one.

The 4096 bit memory will sometimes be referred to as a "word" of the associative memory. PEs with the Tag flip- flops set will be referred to as "selected" PEs, and the associated words as "selected words".

In each Memory Module there is an 8-bit I/O data register used for input and output of data. Usually, data to be put in the associative memory is in the form of vectors, where each item of the vector is to be stored in one memory word. The memory words can only input or output one bit of data at a time, but this may be done in all the 128 words simultaneously. The 128 I/O data registers serve the purpose of transforming the data format between 8 bits in parallel as seen by the Host and 128 bits in parallel in the orthogonal direction.

Thus, the data input process (or equivalently, the data output process) can be divided into two phases: one to fill the I/O data registers from the Host computer or an I/O processor, one to shift the contents into the associative memory. The first phase is totally dominating in time, since the 8-bit data words come in sequentially.

In the implementation made, where no I/O processor is used, the 128 I/O data registers are mapped into the address space of the Host processor. The Host processor can read from and write into these registers in the same way as it reads and writes the cells of its own memory. With each group of eight Memory Modules and Processing Elements there is an address decoding logic that decodes the address information from the Host and produces Read and Write signals for the individual I/O data registers.

A provision is made for writing the same data into all memory words simultaneously. This data is put in the I/O Buffer Register of the Control Unit (see section 1.3.4) and an "I/O Write All" signal is issued by the Control Unit. This places the data in all I/O data registers. Then the contents of these registers are written into the memory words, either to all words, using the "Write All" command, or to selected words only, using the "Write" command.

To read out the contents of a selected memory word, the Host computer (or I/O processor) does not need to know the address of the word. The control signal "I/O Read Selected" puts the outputs of all I/O data registers, except the selected one, in a high-impedance state. The contents of the selected register will be put in the I/O Buffer Register, from where it may be read by the Host processor. If more than one Tag is set, the words could be read out in sequence using a series of "Select First" and Tag-manipulating instructions.

1.3.3 Interconnection structure

All the one-bit outputs from the 128 Memory Modules of LUCAS are gathered on a 128-bit wide bus, accessible from all PEs over an interconnection network. The interconnection network has eight outputs to each PE as shown in Figure 1.3.5. A Data Selector (DS) in each PE is used to choose one of these, according to a Data Select code received from the Control Unit. Most of the interconnection network is changeable through a strapping area for each PE; the straight connection, connecting each PE with its own memory, is the only one fixed.

General purpose interconnection network

Interconnection structures suitable for specific applications can be wired on LUCAS. Among the goals set up for the project was to see which interconnection patterns were suitable for which calculations. However, there is also a need for a general purpose network, capable of permuting data in a flexible way, hopefully useful in many application areas. The study of interconnection networks is a very vivid research area. Many important results have been achieved and several fundamental questions have been raised and not yet solved.

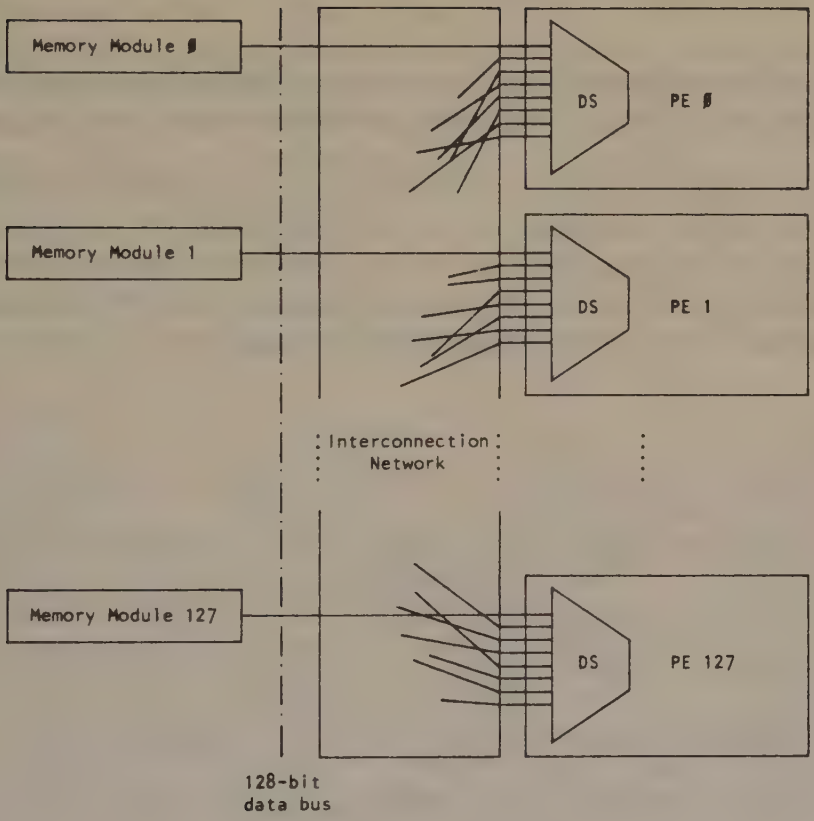


Figure 1.3.5 Interconnection structure.

The STARAN computer uses a network - called the flip network (Batcher 1976, 1977) - that consists of several stages, n stages for an array of $N=2^n$ processing elements. Assuming the processing elements are numbered by binary numbers, level i of the network permits processing element no.

$$P = \langle p_{n-1}, p_{n-2}, \cdots, p_i, \cdots, p_1, p_0 \rangle$$

to exchange data with processing element no.

$$\langle p_{n-1}, p_{n-2}, \cdots, p_i', \cdots, p_1, p_0 \rangle,$$

i.e. with the PE whose binary representation differs from that of P in the i:th position only. This is illustrated in Figure 1.3.6 for n=3.

To control the network, an n-bit vector is used. The i:th element of the vector determines if, on the i:th stage of the network, exchange of data shall be made or not. This control scheme is called "individual stage control". The hardware implementation of the network requires $2^{(n-1)}$ 2-to-1 multiplexers in each stage.

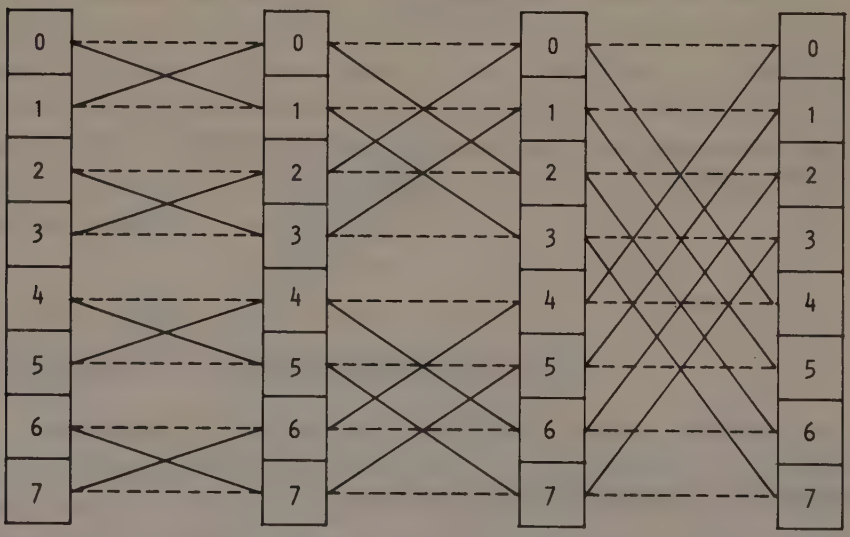


Figure 1.3.6 The Flip network of STARAN.

In addition to the flip network, STARAN is equipped with a shift network, to shift data an arbitrary number of steps up or down.

On LUCAS, a multistage network can not be implemented directly. However, it can be realized in the form of a recirculating network, i.e. using n passes through a one-stage network instead of one pass through an n-stage network. The one-stage network must be able to imitate each of the n stages. Since LUCAS has 128 PEs, seven different connection patterns are required, which is exactly what is available if the input that we normally use for the X flip-flop also is available. Thus, the STARAN flip structure can be implemented on LUCAS. The inputs to each PE are shown in Figure 1.3.7.

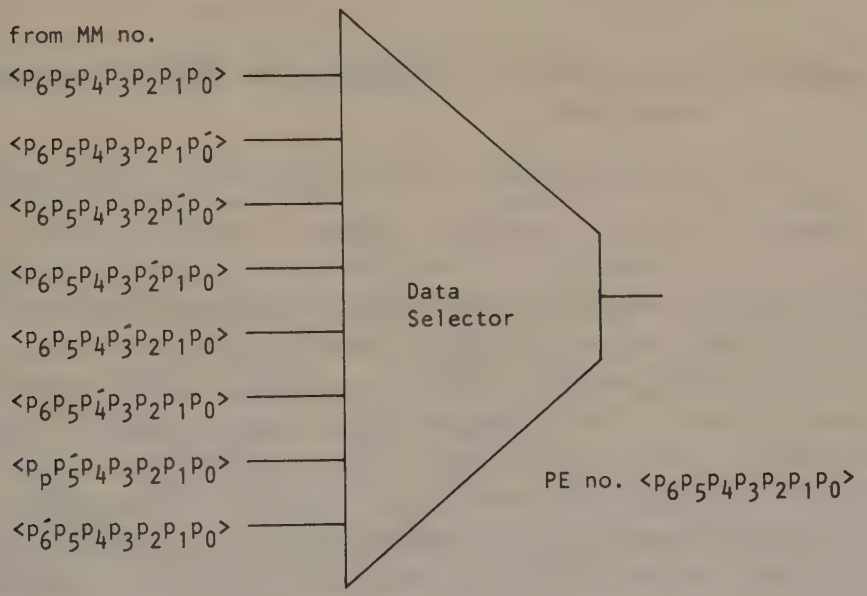


Figure 1.3.7 Data inputs to a LUCAS PE in order to implement the STARAN Flip network.

A convenient way of drawing interconnection networks is by means of interchange boxes (Figure 1.3.8). These are switches with two data inputs and two data outputs. Incoming signals may either pass straight through the box, as in (a), or be interchanged with each other as in (b), depending on the state of the box - "straight" or "exchange".

In order to use interchange boxes to illustrate the flip network, the network must be re-drawn. The nodes within a stage are reordered in a way that makes interchanges occur between neighbours (Figure 1.3.9). This makes it possible to use interchange boxes as shown in Figure 1.3.10.

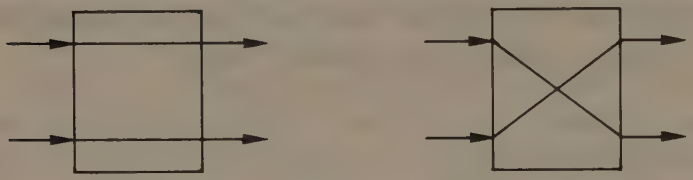


Figure 1.3.8 Interchange box in straight state (left) and exchange state (right).

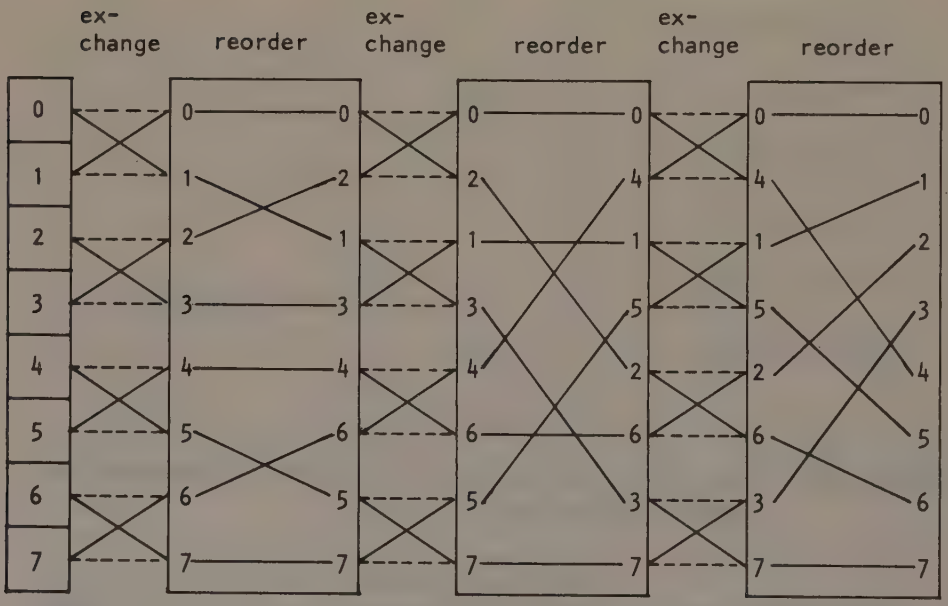


Figure 1.3.9 The flip network re-drawn, showing re-ordering of nodes.

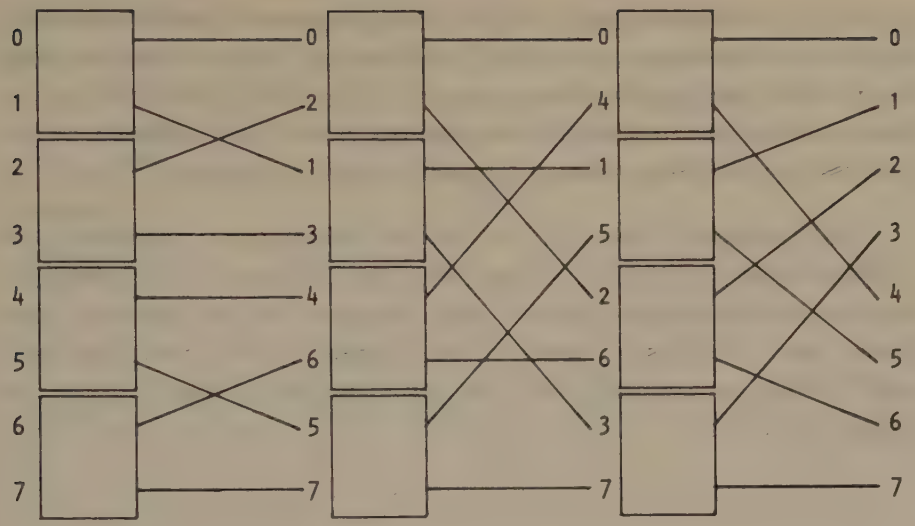


Figure 1.3.10 The flip network drawn using interchange boxes.

When all interchange boxes are in the straight state, data is in the same order on the output side as on the input side. When all boxes are in the exchange state, input 0 will appear at output node 7, input 1 at output node 6, input 2 at output node 5, etc. The number of possible permutations in the flip network is 2^n, since all boxes in a stage are in the same state. Pease (1977) proposes another interconnection network called "the indirect binary n-cube". Its topology is the same as that of the flip network. However, Pease uses individual control of the boxes within a stage.

We will refer to the two methods of switch setting as "individual stage control" and "individual box control", respectively. Individual box control increases the number of possible permutations quite dramatically. For a given input-output pair there is one and only one path that connects them (Pease, 1977). Thus, two different switch settings cannot yield identical permutations. Therefore, the number of

switch settings is the same as the number of permutations that can be performed. There are n levels of switches and N/2 switches at each level. Thus, the network can perform $(2^{N/2})^n = 2^{nN/2}$ different permutations. For n=4, i.e. 16 processors, this is 2^{32}. With 128 processors (n=7) the number of possible permutations is 2^{448}.

The drawback of the flip and n-cube networks as candidates for implementation on LUCAS is that they use different interconnection schemes in each stage. However, the structure can be re-drawn once more, again reordering nodes in all stages but the first and last ones, to obtain identical structures in each stage. This is shown in Figure 1.3.11.

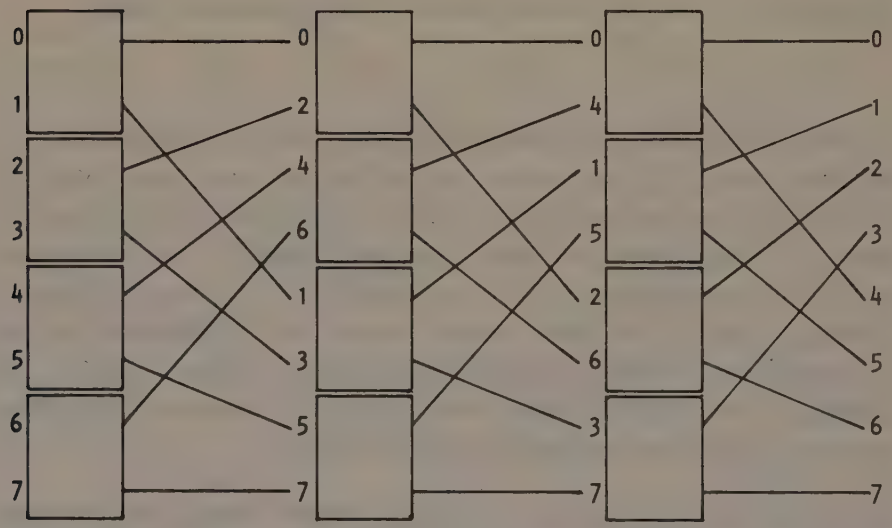


Figure 1.3.11 The flip network re-drawn as n stages of exchange+unshuffle.

The reordering between each stage in Figure 1.3.11 is known as an unshuffle.

This network run backwards, is the omega network proposed by Lawrie (1975), shown in Figure 1.3.12. It consists of two kinds of permutations - the perfect shuffle and the exchange. The name "perfect shuffle" originally describes the process of shuffling a deck of cards perfectly, i.e. intermixing the two halves of the deck perfectly.

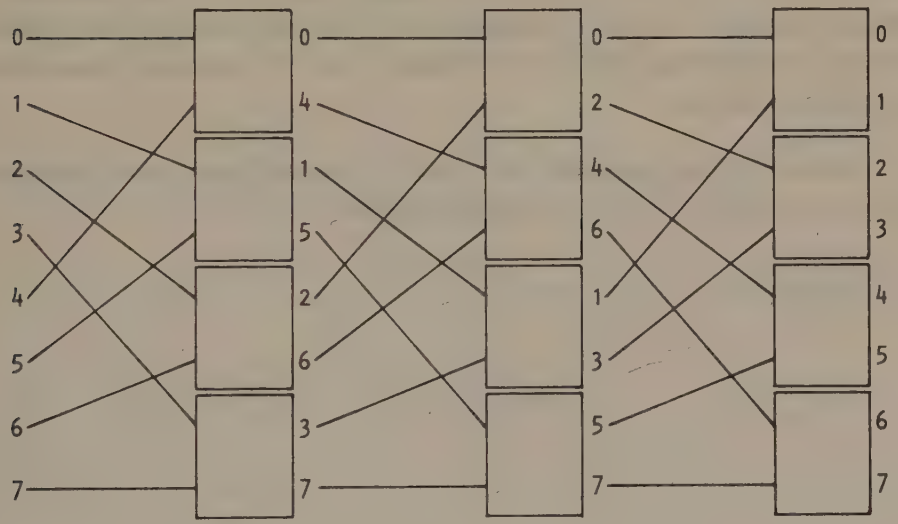


Figure 1.3.12 Lawrie's omega network.

For obvious reasons, this network can perform as many permutations as can its inverse, the n-stage exchange+unshuffle network. However, the sets of possible permutations are not equal if individual box control is permitted. For example, in Figure 1.3.11 input 7 can not be sent to output 2 at the same time as input 6 is sent to output 0, but in Figure 1.3.12 it can. For the transfer 2 to 7 and 0 to 6 the case is the contrary.

As a general purpose interconnection network for LUCAS, the

perfect shuffle+exchange structure has been chosen. The main reasons are: (1) the generality of the network, (2) the few connections needed - only two per PE - and (3) the efficiency of the network for calculation of the Fast Fourier Transform (as shown by Pease(1968) and Stone (1971)), which is useful in both signal and image processing.

The Data Selector of each PE has two inputs from this network - one 'shuffle' input and one 'shuffle+exchange' input - as shown in Figure 1.3.13. If the S input of the Data Selector is used a perfect shuffle is made. If the N (=Neighbour's shuffle) input is used, both perfect shuffle and exchange are made. Thus the exchange operation takes no extra time.

Using the Tag flip-flops as a mask, individual box control may be used on LUCAS. This will be described later.

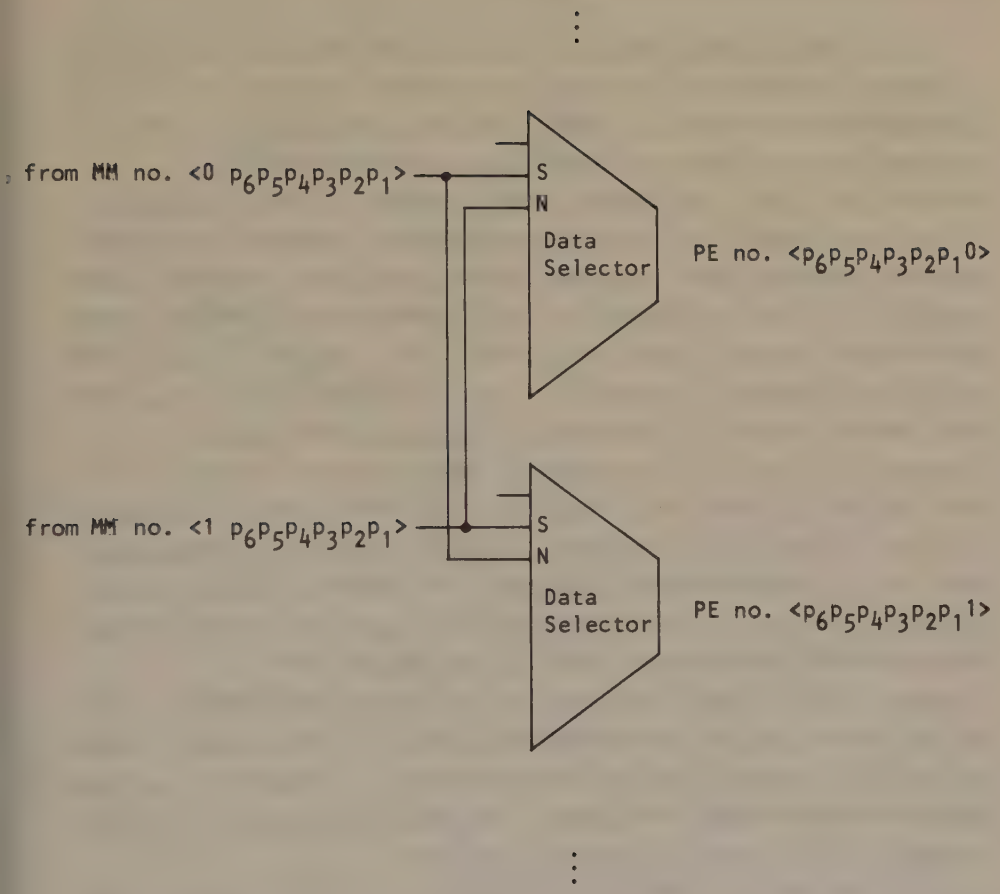


Figure 1.3.13 Connections to Data Selectors for the realisation of the Perfect Shuffle+Exchange network.

As is easily seen in Figure 1.3.12, the perfect shuffle permutation connects Memory Module no. $P=\langle p_{n-1},p_{n-2},\cdots,p_0\rangle$ to PE no. 2*P if P belongs to the first half of the modules and to PE no. 2*P+1 modulo 2^n if P belongs to the second half. This is equivalent to a rotation of the bits in the binary representation of P one step to the left, as indicated in Figure 1.3.13. From this it is clear that n shuffles bring

the elements back in original order.

The perfect shuffle+exchange structure with individual box control is generally regarded as the most useful general purpose interconnection scheme for SIMD computers, since it is capable of realizing the most frequently used permutations ((Stone,1971), (Lawrie,1976), (Yew and Lawrie, 1981)). In spite of this, LUCAS is - to our knowledge - the first implemented SIMD computer to use this network. STARAN uses a similar network, but with individual stage control only, and in addition to this a shift network. PROPAL, from the French company Cimsa (Cimsa,1979), has between 8 and 2048 processing elements organized in a ring-connected linear array with only nearest neighbour communication, but via a 16-bit wide connection. Illiac IV (Barnes et al., 1968) with its 64 PEs and DAP with its 1024 or 4096 PEs (Flanders et al.,1977) both use a network with four inputs per PE, permitting access to the data of neighbouring PEs in a two-dimensional structure. Processor arrays intended especially for image processing have also used the two-dimensional structure with communication with four neighbours (MPP, (Batcher,1980)) or with eight neighbours (CLIP4, (Duff,1979)). LUCAS may be wired for this interconnection structure when used in image processing. However, as will be described in Chapter 5, we use another storage scheme for images, requiring communication only with neighbours in a one-dimensional arrangement.

Thus, in the network presently used in LUCAS, we have also included data paths from the neighbours above and below, as shown in Figure 1.3.14. One input to the Data Selector is the output from the X-register of the PE.

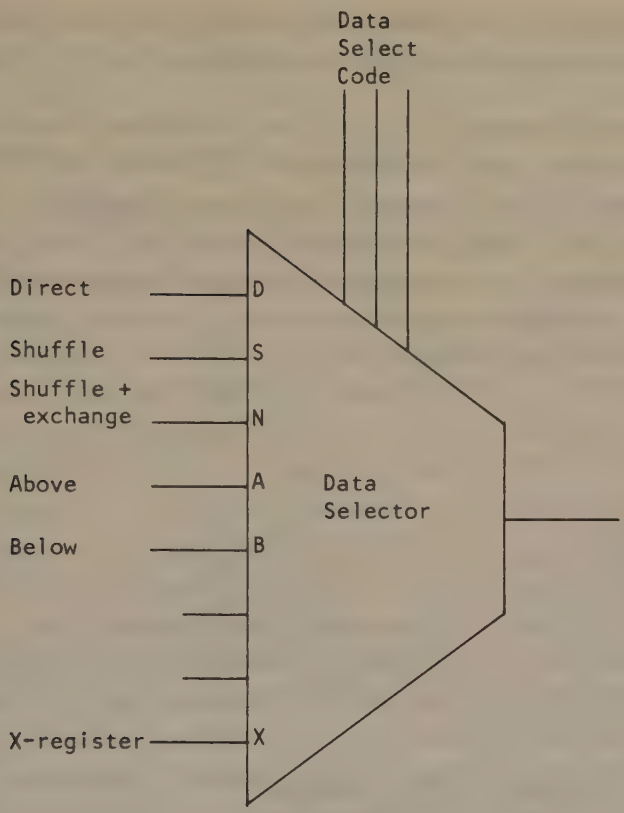


Figure 1.3.14 Implemented connections to the Data Selector of the Processing Element.

All Data Selectors receive the same Data Select code from the Control Unit. Therefore, there is no direct possibility to implement individual box control, because this requires that some Data Selectors choose the Shuffle input, and some the shuffle+exchange input. This has to be done in two steps, using the state of the Tag flip-flop to decide the box setting. In the first step, the R flip-flops of all words are loaded from memory via the Shuffle input. In the second step, the

R flip-flops are again loaded from memory, but only in selected words, and now using the Shuffle+exchange input.

In the omega network of Lawrie the interchange boxes have two additional settings, also implementable on LUCAS as a two-step process, namely "upper broadcast" and "lower broadcast", shown in Figure 1.3.15. This opens possibilities to broadcast a few data items to many processors in a small number of steps. For example, one data item may be broadcast to all processors in n steps (n is the number of stages in the omega network).

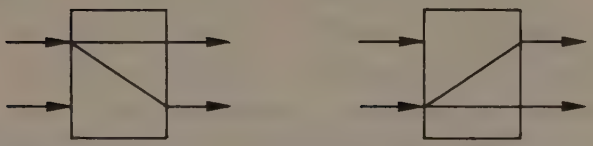


Figure 1.3.15 Broadcast switch settings.

There is an additional means of data broadcasting on LUCAS, probably more useful than the one mentioned above in applications where the content addressability of the memory is used. This is the possibility to select one word, read out the contents to the I/O Buffer Register, then select another set of words and write the contents of the I/O Buffer Register into these words. This procedure takes four clock periods for each bit to be moved, regardless of the number of destinations. A major advantage is that the addresses of the destination words need not be known. It has proved very practical in the manipulation of data base relations.

In conclusion, the interprocessor communication facilities hitherto implemented and used on LUCAS are of three types:

* A perfect shuffle+exchange network

* Nearest neighbour communication in one dimension

* Communication one-to-many via the I/O bus

As mentioned, the latter has proved useful in data base applications. Nearest Neighbour communication is essential in image processing. We will discuss extensions of it in chapter 5. The perfect shuffle+exchange network is perfectly suited for the Fast Fourier Transform, which is of great importance in signal processing. This network also has the ability of performing any permutation, if recirculated the required number of times. The upper limit of this number is still unknown, as is a procedure to determine the optimal switch settings. Parker (1980) has shown that $3 * \log_2 N$ passes (N is the number of PEs) is always enough to perform any permutation (i.e. one-to-one mapping) and that $6 * \log_2 N$ passes is enough for any mapping (one-to-many mappings of data items are allowed). No one has shown a permutation that cannot be implemented in $2 * \log_2 N$ passes. Parker suspects that $2 * \log_2 N$ passes may always be enough for permutations.

Lawrie(1975) has devised a very simple method to decide the switch settings for the implementation of those permutations that can be performed in one pass through the omega network (i.e. in $\log_2 N$ passes through the network of LUCAS). An extension of the algorithm to include the control of two passes through the omega network is given in (Yew and Lawrie, 1981). It is also shown that the permutations realizable using this algorithm include all permutations stated by Lenfant (1978) as the most frequently used in parallel processing. Examples of the use of these algorithms will be given in chapter 2.

1.3.4 Control Unit

The task of the Control Unit (Figure 1.3.16) is to receive instructions from the Host computer and interpret them in the form of sequences of control signals sent to the Processor Array where the actual processing takes place. An instruction consists of an operation code and a number of parameters. The latter specify the addresses of source and destination data, the lengths of data items and the sizes of arrays (e.g. the number of columns in a matrix). The Control Unit receives the operation code in the Instruction Register (IR) and the parameters in four Parameter Registers (PR1..PR4).

Common and Mask Registers

The Control Unit may also contain data to participate in the execution of instructions. These are held in the Common and Mask Registers, which are the same kind of memory modules as those used in the Processor Array. For input/output they are each equipped with an 8-bit I/O shift register.

In some instructions, one argument is to be used in the computations of all PEs. That argument is then conveniently placed in the Common Register. The one-bit output of the Common Register reaches all PEs.

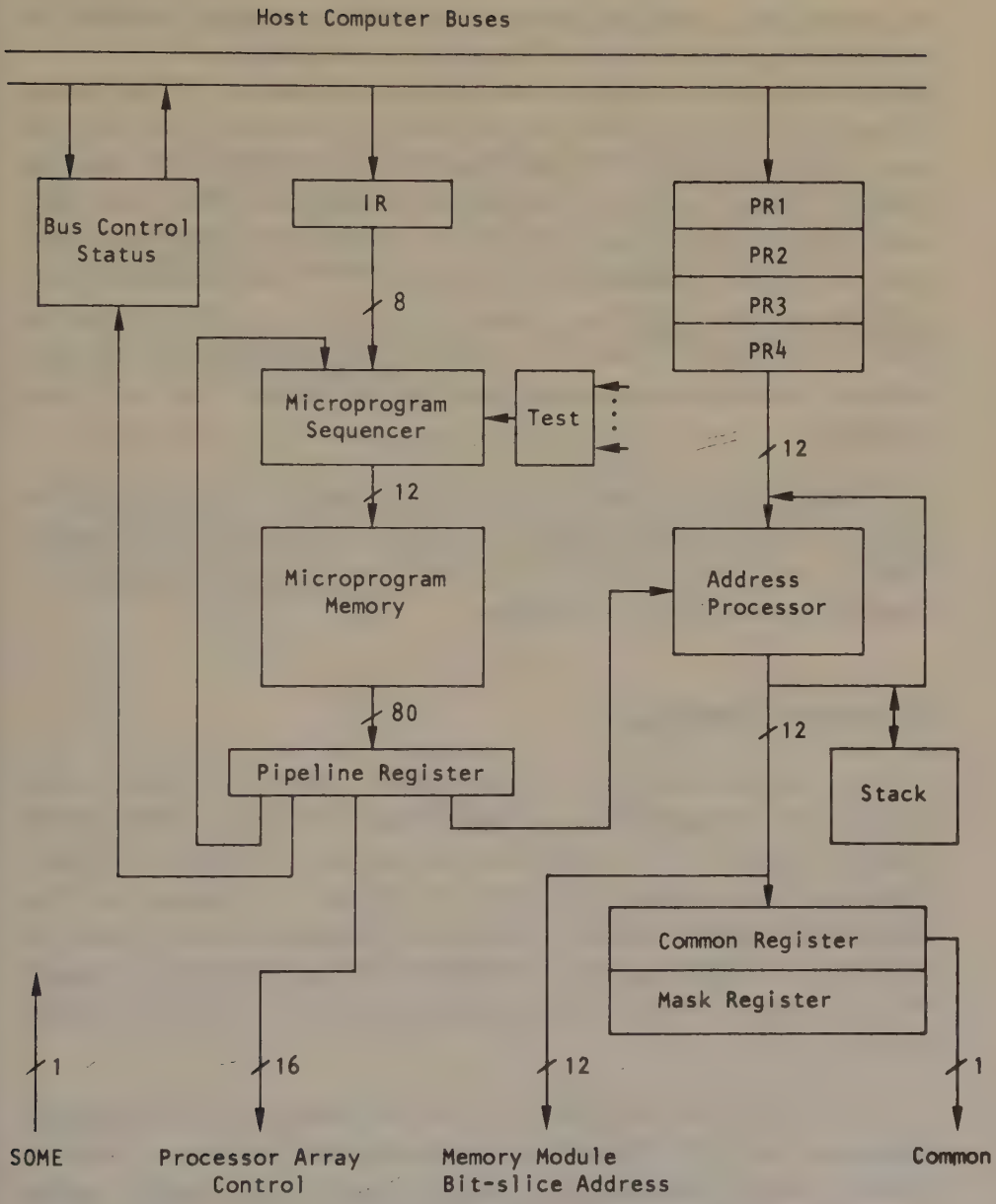


Figure 1.3.16 LUCAS Control Unit

The presence of a mask register in LUCAS was one of the subjects of discussion during the design phase. The traditional use of a mask register in associative processors is the following: We want

to operate on a field of the array, but skip over some bit-slices of the field. In implementations of associative processors with bit-parallel working mode the mask register is essential. In bit-serial implementations like LUCAS, the selection of some bit-slices but not others is done by means of field specifications. However, it might be useful in some applications to be able to use both ways. The Mask Register can also be used to decide which of two different operations that are to be performed on a bit-slice. In some operations, e.g. multiplication of a field by a scalar, one of the arguments is put in the Mask Register. When this argument is scanned, bit by bit, different actions are performed depending on whether a zero or a one is read.

Called a mask register or not, the presence of a register of the same kind as the memory words and with a possibility for the Control Unit to use the output of it to determine which path to take, is essential in this kind of processor.

Microprogram Sequencer

The interpretation of an instruction takes the form of the execution of a microprogram. A microprogram memory of 4096 80-bit words is used. It is of read/write type and can be loaded (and read) from the Host computer over its I/O system. This means that the microprogram memory can be accessed in two modes: from the Host as 40960 8-bit words, and from the Control Unit as 4096 80-bit words. At the outputs of the microprogram memory is an 80-bit pipeline register.

To control the sequence of microinstructions a Microprogram Sequencer, Am 2910, (Figure 1.3.17) is used (Mick and Brick, 1980). It permits stepping through a microprogram line by line, jumping, calling subroutines, conditionally repeating instructions, etc. To do this it has an instruction set of 16 instructions, given in Table 1.3.2. The main inputs of Am 2910 are

* 4 bits specifying the sequencer function to be performed.

* 1 bit that is tested upon by conditional instructions.

* 12 bits specifying the next microprogram address in jump instructions. These inputs are also used to load the internal counter and to transfer the contents of the Instruction Register to the Microprogram Sequencer.

The major output of Am 2910 is a 12-bit address used as address pointer to the microprogram memory.

The test input of the Microprogram Sequencer gets its value from the output of an 8 in/1 out multiplexer. The value may be inverted by a control signal, giving a total of 16 possible test conditions. We will return to these later. In addition, the internal counter of Am 2910 can be tested for zero value.

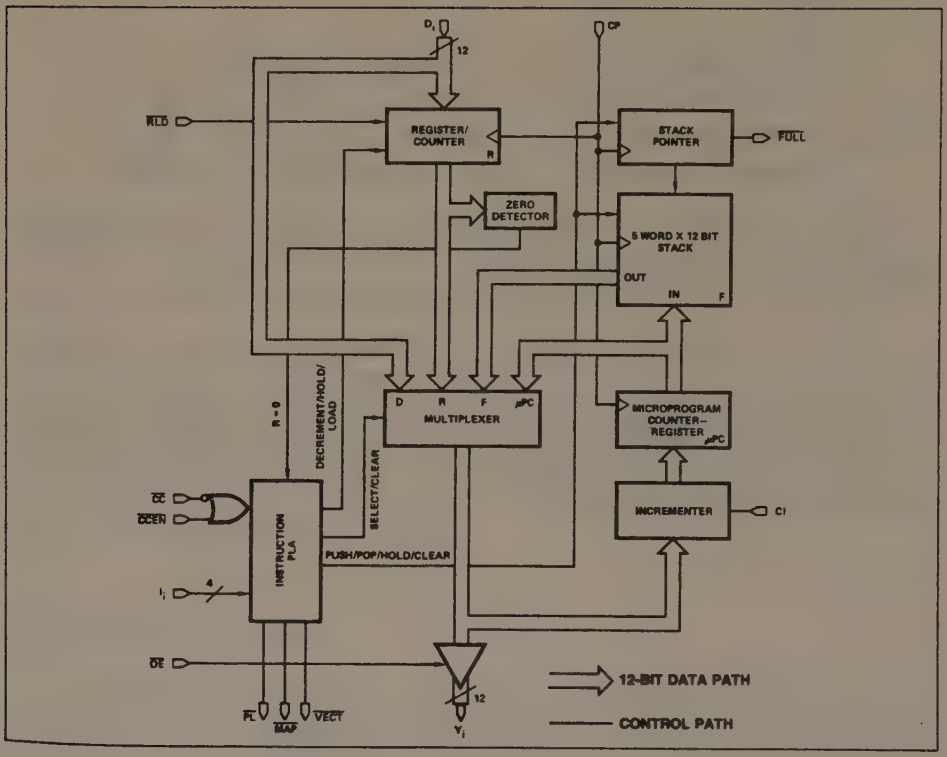


Figure 1.3.17 Am 2910 Microprogram Sequencer. (Reprinted from (Mick and Brick, 1980))

CONT CONTINUE. The next microinstruction is taken.

JZ JUMP ZERO. Jump to location zero in the microprogram memory.

JMAP JUMP MAP. Jump to the address in the Instruction Register.

CJP CONDITIONAL JUMP. Jump to PL if the condition is true. Else continue.

CJV CONDITIONAL JUMP VECTOR. Jump to the address in the I/O Buffer Register if the condition is true.

JRP JUMP R/PL. If the test fails jump to the address in the internal Register/Counter. Else jump to PL.

CJS CONDITIONAL JUMP TO SUBROUTINE. Subroutine call to PL if the condition is true. Else continue.

JSRP JUMP TO SUBROUTINE R/PL. If the test fails perform a subroutine call to the address in the internal Register/Counter. Else perform a subroutine call to PL.

CRTN CONDITIONAL RETURN FROM SUBROUTINE. If the condition is true take next address from stack and pop stack. Else continue.

PUSH PUSH/CONDITIONAL LOAD COUNTER. Continue and push next address on stack. If the condition is true load PL into the internal counter.

RFCT REPEAT LOOP IF COUNTER NOT ZERO. Used to repeat a loop which has been set up with the PUSH instruction. If the counter is nonzero it is decremented and a jump to the top address of the stack is performed. Otherwise continue and pop the stack.

LOOP TEST END LOOP. An alternative possibility to repeat a loop set up with the PUSH instruction. The Test input is used as condition. When false the branch back is performed. When true continue is performed.

CJPP CONDITIONAL JUMP PL AND POP. Used to exit from inside loops set up with the PUSH instruction. Jump to PL and pop stack if the Test input is true, else continue.

LDCT LOAD COUNTER. Load PL into the internal Register/Counter.

RPCT REPEAT PL IF COUNTER NOT ZERO. Similar to RFCT but the address is taken from PL rather than from the stack top. Used to terminate loops set up with the LDCT instruction.

TWB THREE WAY BRANCH. Either continue, jump to PL or jump to the address at the stack top, depending on the Test input and the counter value.

Table 1.3.2 Am 2910 instruction set

Address Processor

For efficient processing of data in the Array it is important that the Control Unit is capable of simultaneously outputting addresses to bit-slices in the Memory Modules at high speed and counting the number of loops in the microprogram. The part of the Control Unit that handles this is the Address Processor. The Address Processor is controlled by fields in the microinstructions to output correct sequences of addresses and provide loop counting services. It is built around three 4-bit arithmetic units of type Am 2901A, which comprise a 12-bit processor with 16 internal registers capable of performing integer addition and subtraction, incrementation and decrementation, etc. Attached to this processor is a pushdown stack memory of 32 12-bit words onto which addresses can be stored and retrieved during computation. Also included in the Address Processor is a separate 12-bit loop counter, thus permitting loop counting and address calculation to be performed simultaneously. Finally, the Address Processor contains an Address Register that holds the address currently in use, while the next one is being calculated.

Data inputs to the Address Processor are: the Parameter Registers PR1..PR4, 12 bits of the microinstruction, and the I/O Buffer Register (see below). Two of the 16 internal registers can be accessed simultaneously through two 4-bit addresses, A and B. The register accessed through the A-address (referred to as register (A)) can only be read from, while the one accessed through the B-address (register (B)) can both be read from and written into.

In addition to these data and address inputs, 11 bits are used to control the working of the Address Processor: 4 bits determine the function to be performed by the arithmetic unit, 3 bits determine the input to be used, two bits control loading and decrementation of the loop counter, and two bits control the working of the pushdown stack.

The 16 functions of the arithmetic unit are the following:

```
0)  TBZ      output = reg(B)

1)  LDBD     reg(B):= input
             output = input

2)  LDBA     reg(B):= reg(A)
             output = reg(A)

3)  INCB     reg(B):= reg(B)+1
             output = reg(B)+1

4)  DECB     reg(B):= reg(B)-1
             output = reg(B)-1

5)  ADAD     reg(B):= reg(A)+input
             output = reg(A)+input

6)  TAEQB    output = reg(B)-reg(A)

7)  ADBA     reg(B):= reg(B)+reg(A)
             output = reg(B)+reg(A)

8)  SUBBA    reg(B):= reg(B)-reg(A)
             output = reg(B)-reg(A)

9)  AINCB    reg(B):= reg(B)+1
             output = reg(A)

10) ADECB    reg(B):= reg(B)-1
             output = reg(A)

11) AADAD    reg(B):= reg(A)+input
             output = reg(A)

12) AADBA    reg(B):= reg(B)+reg(A)
             output = reg(A)

13) ASUBBA   reg(B):= reg(B)-reg(A)
             output = reg(A)

14) ALDBD    reg(B):= input
             output = reg(A)

15) PLADR    output = input
```

Control of the Processor Array

Of the 80 bits in the microinstruction that each clock cycle appears at the output of the Pipeline Register, 16 bits are used to control the Processing Elements, the I/O data registers and writing in the Memory Modules. These 16 bits can be grouped as follows.

5 bits determine the function code sent to the ALUs of the Processing Elements.

3 bits control the Data Selectors of the PEs.

1 bit is the Select First command.

2 bits control writing in the Memory Modules: Write All and Write Tagmasked, respectively.

1 bit determines the source of the data that is written into the Memory Modules: the R-register contents or a bit from the I/O data register.

3 bits control the I/O data registers of the Memory Modules (shift control, read and write control).

1 bit enables the buffer outputting one bit from each Memory Module to the 128-bit wide data bus.

Conditions available for test

The value of one input (CC) to the Microprogram Sequencer determines if a specified test condition is fulfilled or not. Through a Test Multiplexer different signals can be gated to this input. These signals are

ZAP: Zero Address Processor. If the output of the arithmetic unit of the Address Processor was zero after the previous clock

cycle, this signal will have the value 'one'.

MASK: The data output of the Mask Register.

NZLC: Non-Zero Loop Counter. If the output of the loop counter in the Address Processor was non-zero after the previous clock cycle, this signal will have the value 'one'.

IL: Instruction Loaded. The output of the BUSY flip-flop which is automatically set when the Instruction Register is loaded (see description of Host interface below).

NONE: A signal from the Processor Array. If no Tag flip-flops are set this signal will have the value 'one'.

TRUE: This signal always has the value 'one'. It is used to make conditional instructions unconditional.

Through an additional control signal the output of the Test Multiplexer can be complemented. Therefore, the following twelve test conditions are available:

ZAP, NZAP, MASK, NMASK, NZLC, ZLC, IL, NIL, NONE, SOME, TRUE, FALSE.

In addition to the tests available through the Test Multiplexer, the internal counter of Am 2910 can be tested for zero contents. This test is implicit in some instructions.

1.3.5 Host Computer Interface

It is the duty of the Host computer to supply the Control Unit with instructions and the Processor Array with data for computation, and also receive output data from the array. In order to carry out this task properly the Host computer must have access to information about the status of the ongoing computation.

The interface between the Host computer and the parallel processor has intentionally been kept very simple. This facilitates exchange of Host computer. All communication is done via registers accessible both from the Host and from the parallel processor. In most cases one of the parties has exclusive right to write, while the other has exclusive right to read the register. However, for data input and output there are registers that can be read from and written into by both parties.

Through address decoding circuitry in the Control Unit, all the registers that constitute the interface are mapped into the memory address space of the Host computer. This means that the Host computer's "image" of the parallel processor simply is a memory area.

The Control Unit is connected to the address, data and control buses of the Host computer. To issue an instruction to LUCAS the Host computer writes the parameters and the operation code in specific memory addresses. The address decoding circuitry thereby generates load signals for the Parameter Registers and the Instruction Register. By the latter signal, a BUSY flip-flop is set, preventing new parameters and operation codes from being loaded. The flip-flop is cleared by a control bit in the microinstruction when the operation code and parameters have been loaded into the Microprogram Sequencer and the Address Processor, respectively.

The I/O data registers, present in all Memory Modules, are likewise mapped into the address space of the Host computer. The Host can both read and write in these registers. A write command can also be issued by the Control Unit, in the form of the signal "I/O Write All". The data residing in an "I/O Buffer Register" of the

Control Unit is then loaded into all the I/O data registers. The I/O Buffer Register also has a fixed address in the Host address space. This provides the Host computer with a possibility to write the same data into all Memory Modules without sending the same data 128 times. Furthermore, the Control Unit can read the I/O data registers by the control signal "I/O Read Selected". On this command, data from that I/O data register where the corresponding PE has its Tag flip-flop set is transferred to the I/O Buffer Register. Finally, the Control Unit can order transfer of data in another format to and from the I/O data registers, namely the information in the associated Memory Modules. This information is transferred bit-serially in each word, all words in parallel.

The Host computer may any time interrogate a Status Register in the Control Unit. The Host has only read access to this register. The register is loaded under the control of a bit in the microinstruction. The status information given contains the following six bits:

BUSY: Indicates that the Instruction Register or the Parameter Registers contain information not yet transferred to the Microprogram Sequencer and Address Processor, respectively.

SOME: Indicates that at least one of the PEs has a 'one' in its Tag flip-flop.

ZAP: Zero Address Processor. Indicates that the output of the arithmetic unit in the address processor has zero value.

ZLOOPC: Zero Loop Counter. Indicates that the output from the loop counter in the Address Processor has zero value.

S1, S2: Two bits of the microinstruction that are not used for control purposes and thus can be used for any kind of signalling from the Control Unit to the Host, e.g. interrupt.

1.3.6 Physical Description

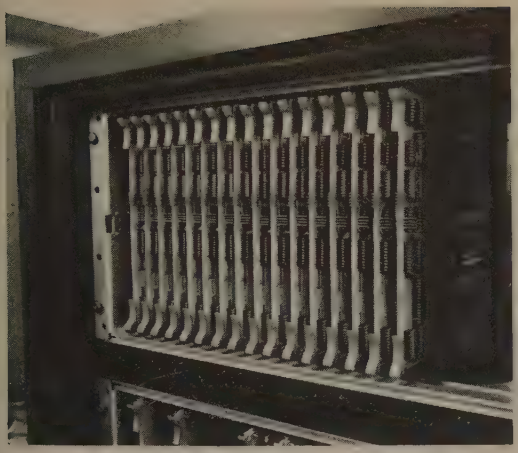
The Processor Array, consisting of 128 Processing Elements with associated Memory Modules and input/output circuitry, occupies 16 printed circuit boards (Photo 1.3.1 (a)). Each board of eight processors measures 220 x 230 mm (Photo 1.3.1 (b)). It contains 70 IC packages, 59 of which implement the processors and 11 of which are signal buffers and I/O address decoding circuitry. Detailed drawings of a Processor Board are shown in the appendix.

The Processor Boards are mounted in the top rack of a 1600 x 600 x 600 mm cabinet housing LUCAS (Photo 1.3.2). Immediately below resides the Control Unit which is divided into two boards. A third board in this rack contains the circuitry necessary for the loading of microprograms into the Microprogram Memory and also a display of the microinstruction word, which was used for debugging of the hardware.

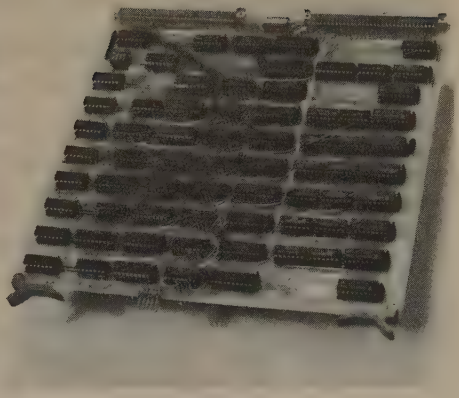
The next rack contains the Host processor. It is a conventional Z80 system built from Prolog STD-bus cards. It also includes a floppy disk unit and an image display system.

The communication between the different racks is via flat cables along one side of the racks.

The Host computer is connected to the department's VAX-11/780 computer via a serial channel. Most of the program development is now done on that system.



(a)



(b)

Photo 1.3.1 (a) The 16 Processor Boards
(b) A Processor Board comprising eight Processors

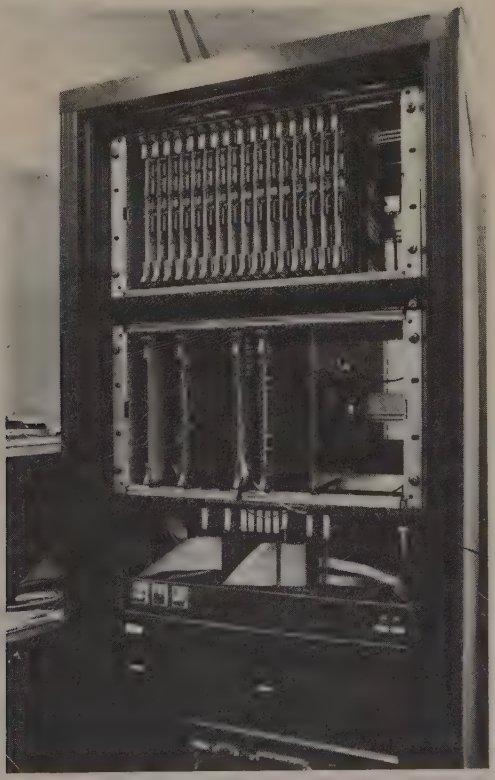


Photo 1.3.2 LUCAS

1.4 COMPARISON WITH OTHER, RELATED DESIGNS

A few machines with characteristics similar to those of LUCAS have been built. We will review the main features of six such designs and compare them to LUCAS. Two of the machines have been implemented in a university environment (Vastor and CLIP4). Three of them (STARAN, DAP and PROPAL 2) are commercial products, but very few copies of them have been built. MPP finally, is built by industry for dedicated use by NASA. STARAN, DAP, PROPAL 2 and Vastor can be considered as being general purpose processor arrays, whereas CLIP4 and MPP are designed exclusively for image processing.

1.4.1 STARAN

STARAN (Batcher, 1974) from Goodyear Aerospace Corporation was the first bit-serial processor array to be built. The first machine was demonstrated in 1972. It was built from standard off-the-shelf integrated circuits.

An array in STARAN consists of 256 Processing Elements. Several arrays can be used together. Each Processing Element has three flip-flops. Any Boolean function of one bit from the memory and one bit from either of the flip-flops can be performed. The memory words are 256 bits long, in later versions extended up to 9 kbits.

A key element of the STARAN computer is the multidimensional access (MDA) store (Batcher, 1977). Elements of data are scrambled in a certain way as they are written into memory so that data can be accessed in different directions. For example, a 256 bit word or a 256 bit slice are accessed equally easy. This is achieved by always putting two bits of the same word in different memory chips and at different addresses. Therefore, modifications of the bit-addresses to each memory chip, and passing of data through a so called Flip network, are needed.

The Flip network also accounts for the routing of data between Processing Elements (see Section 1.3.3). It has many characteristics in common with the perfect shuffle/exchange network implemented on LUCAS. The STARAN network is of multistage type, whereas LUCAS uses a single-stage network. Individual box control, as discussed in Section 1.3.3 is difficult to achieve on STARAN but can be programmed on LUCAS as shown in Section 2.2.3.

Input/output takes place over a 32-bit wide data bus. The Flip network allows different subsets of the PEs to be reached in one I/O operation. Also, the MDA store allows the 32 bits to be stored in parallel in one word. A provision is made also for 256 bit parallel input/output, but this requires special purpose peripheral configurations.

In conclusion, STARAN and LUCAS have many features in common. The STARAN architecture had great influence on the design of LUCAS.

1.4.2 DAP

The Distributed Array Processor - DAP - (Reddaway, 1979) was developed by the ICL company in England. The first experimental model was operable in 1976. The first customer model was delivered to London University in 1980.

The pilot DAP has 1024 PEs arranged in a 32 x 32 array. The customer model has 4096 PEs in a 64 x 64 configuration. Each PE can communicate with its nearest neighbours in the North, South, East and West directions.

The PEs comprise three one-bit registers: Activity, Carry and Result registers. Associated with each PE is a 4 kbit memory module. All memory modules receive the same bit address; no address modification and permutation of data, as in STARAN, is done.

For the purpose of input/output and data broadcasting, "highways"

in the vertical and horizontal directions are used. The width of a highway is the same as the side of the processor array, thus allowing a PE to send data to all PEs in the same row or in the same column. Thus, DAP allows 32-bit wide I/O transfer in the pilot version and 64-bit wide transfer in the customer version.

A special feature with DAP is that the totality of memory modules forms a standard store module of the Host computer. This is in a way similar to the LUCAS arrangement in which any 8-bit slice of the memory can be mapped into the address space of the Host computer via the I/O registers. However, data is accessed in entirely different ways. While on LUCAS an 8-bit portion of each memory word is accessed at a time, on DAP a 32-bit slice over a row or column of the PE is available. The difference lies in the fact that with LUCAS the Host accesses the I/O data registers which are the medium for data format conversion. With DAP the PEs are accessed directly.

1.4.3 PROPAL 2

PROPAL 2 (Cimsa, 1979) is a processor array computer designed and marketed by the French company CIMSA. The number of processors of a system can be between 8 (one board) and 2048 (8 racks of 32 boards), arranged with one-dimensional connectivity. A "condition" flip-flop is included in each PE together with a 16-bit work register called the "Ascenceur". A Boolean one-bit arithmetic/logic unit takes one bit from the Ascenceur and the other bit from either the condition flip-flop, the associated 256-bit memory word or a parallel input. An 8-bit shift register is included which speeds up multiplication by holding the partial products.

The Ascenceur registers have got their names from their role in interprocessor communication. The contents of a register can be transfered in parallel to the neighbouring PE above or below. End-around connectivity can be used between the top and bottom PEs under control of the Host.

Input and output can be performed in two ways: One is through the Ascenceur registers which act as a 16-bit wide shift register. During the shifting, processing in the PEs cannot be done. The other I/O method uses a one-bit input line of each PE. Thus, the parallelity of this path can be extremely large.

PROPAL 2 is controlled by a microprogrammable minicomputer. The microinstruction word is divided into two halves of 48 bits each. One is used to control the minicomputer itself, the other controls the parallel processor.

1.4.4 Vastor

Unlike all other systems surveyed in this section, the Vastor computer (Loucks et al. 1982) designed at the University of Toronto, Canada, uses a standard, off-the-shelf processing element - the Motorola MC14500B. This has been marketed as a control unit for industrial applications.

The number of PEs is 256. Each PE has a 1024 bit memory word and all memory words receive the same bit address.

A shift register with one bit per PE is the medium used for interword communication and input/output. Passing data between PEs takes considerable time unless mere shifting is required. The conversion from byte oriented to bit-slice oriented data necessary at input can be done by the shift registers, but only in every 8th word at a time. This makes input/output quite complicated.

The most interesting thing about Vastor when comparing it to LUCAS is its size and cost, which both are about the same as for LUCAS. All other machines that we consider represent much larger implementational efforts. In the case of Vastor, 16 boards are used and a component cost of $2000 is reported for a full 256 PE system. (Only 16 PEs have actually been implemented).

Performance results reported for Vastor indicate that LUCAS is about 10 times faster on operations that do not require inter-PE communication and more than a hundred times faster on tasks that require such communication.

1.4.5 CLIP4

CLIP4 (Duff, 1979) is the most recent of a series of processor arrays which have been constructed at University College, London. It is exclusively aimed at image processing. 9216 PEs are arranged as a 96 x 96 array. Each PE communicates with eight neighbours, four of which are diagonal.

32 bits of memory are associated with each PE, which is much less than any other processor that we consider. The PEs contain three one-bit registers each. Two independent Boolean functions of two inputs each can be performed simultaneously.

1.4.6 MPP

The Goodyear Areospace Corporation, who designed and marketed STARAN, was contracted by NASA in 1979 to build a "massively parallel processor", MPP (Batcher, 1982), for the processing of satellite images. It has an architecture similar to DAP with 128 x 128 processing elements connected in a two-dimensional arrangement.

Each PE has associated a 1 kbit memory word. The PEs are slightly more powerful than the DAP PEs. Each one contains five one-bit registers, a full adder and a two-input Boolean logic unit. Furthermore, a variable-length shift register is included to speed up multiplication.

An additional one-bit register, S, used exclusively for data routing, is included in each PE. When inputting data (i.e. images), 128 bits are input in parallel at one edge of the array and stored in the

S-registers. Data is shifted from column to column of S-registers until, finally, the whole image is input. Not until now, the processing is interrupted and the input data stored in the memories. Input and output can proceed simultaneously so that an input image is shifted in at one edge while at the same time an output image is shifted out at the opposite edge.

Thus, the PEs of MPP can be said to be connected in two different ways: For processing a two-dimensional connection is used, whereas for input/output the PEs are connected as 128 separate 128-bit shift registers.

1.4.7 Conclusion

The machines surveyed share one important property: they all comprise a large amount of bit-serially working processing elements. In fact, the processing elements of all the designs are rather similar, although some of them are slightly more powerful than the others. The processing elements of LUCAS fall somewhere near the middle of the scale.

The differences between the various designs mainly concern the interconnection arrangement and the input/output structure. As for PE interconnection, LUCAS is most similar to STARAN, which is the only design that uses a network akin to the perfect shuffle/exchange network. However, LUCAS does not have the complicated memory storage method that STARAN utilizes. The input/output system is also quite different.

LUCAS is similar to DAP and MPP in the simplicity of data storage but different with regard to interconnection and input/output.

The size and cost of LUCAS are like those of Vastor. All other machines are larger and more expensive.

CHAPTER 2
BASIC INSTRUCTIONS

In this chapter we will describe the basic types of instructions that can be performed on data in the Processor Array. Examples of the different instruction types are given and execution times are calculated.

2.1 CLASSIFICATION OF INSTRUCTIONS

At each moment one bit from each memory module is accessible by the PEs. Taken together these bits form a bit-slice of the total memory. The total memory will often be referred to as the associative memory (AM). A number of consecutive bit-slices form a field of the AM. (Since the memory chips are of random access type, the constraint that the bits must be consecutive is not necessary, but this is normally the case). The contents of the Tag flip-flops may be used to mask out certain words in an operation. We use the term selector to refer to the bit vector describing the contents of these flip-flops. A selector may be stored in a one-bit field of the associative memory.

2.1.1 Basic types of instructions on the associative memory

The instructions used to manipulate data in the associative memory may be classified according to the types of operands and the types of results. An operand may either be a constant, stored in the Common Register, or a vector, stored in a field of the associative memory. (We will use 'field' and 'vector' alternatingly to denote this type of operand). The result of an operation is either a field or a selector, i.e a one-bit field.

A selector can always be used in an operation to mask out some PEs. Thus, it may also be regarded as an operand, present in all operation types. We do not explicitly indicate this when we list the different types below.

This view results in the following six basic instruction classes, each one illustrated in Figure 2.1.1.

(a) <field> --> <field>

Examples: Increment field

Copy field

Permute field (the vector in the destination field is a certain permutation of the vector in the source field)

(b) <field> --> <selector>

Examples: Maximum value of a field

Minimum value of a field

(c) <field,field> --> <field>

Examples: Add fields

Multiply fields

Maximum of fields (The elements of the source vectors are compared pairwise and the maximum of them is put in the destination field)

AND between fields

(d) <field,field> --> <selector>

Examples: Field equal to field (The selector will mark those words where the elements of the two fields are identical)

Field larger than field (The selector will mark those words where the elements of source field 1 are larger than the corresponding elements of source field 2)

(e) <constant,field> --> <field>

Examples: Add constant

Multiply by constant

Subtract constant

Subtract from constant

AND with constant

(f) <constant,field> --> <selector>

Examples: Exact match

Closest match

Greater than constant

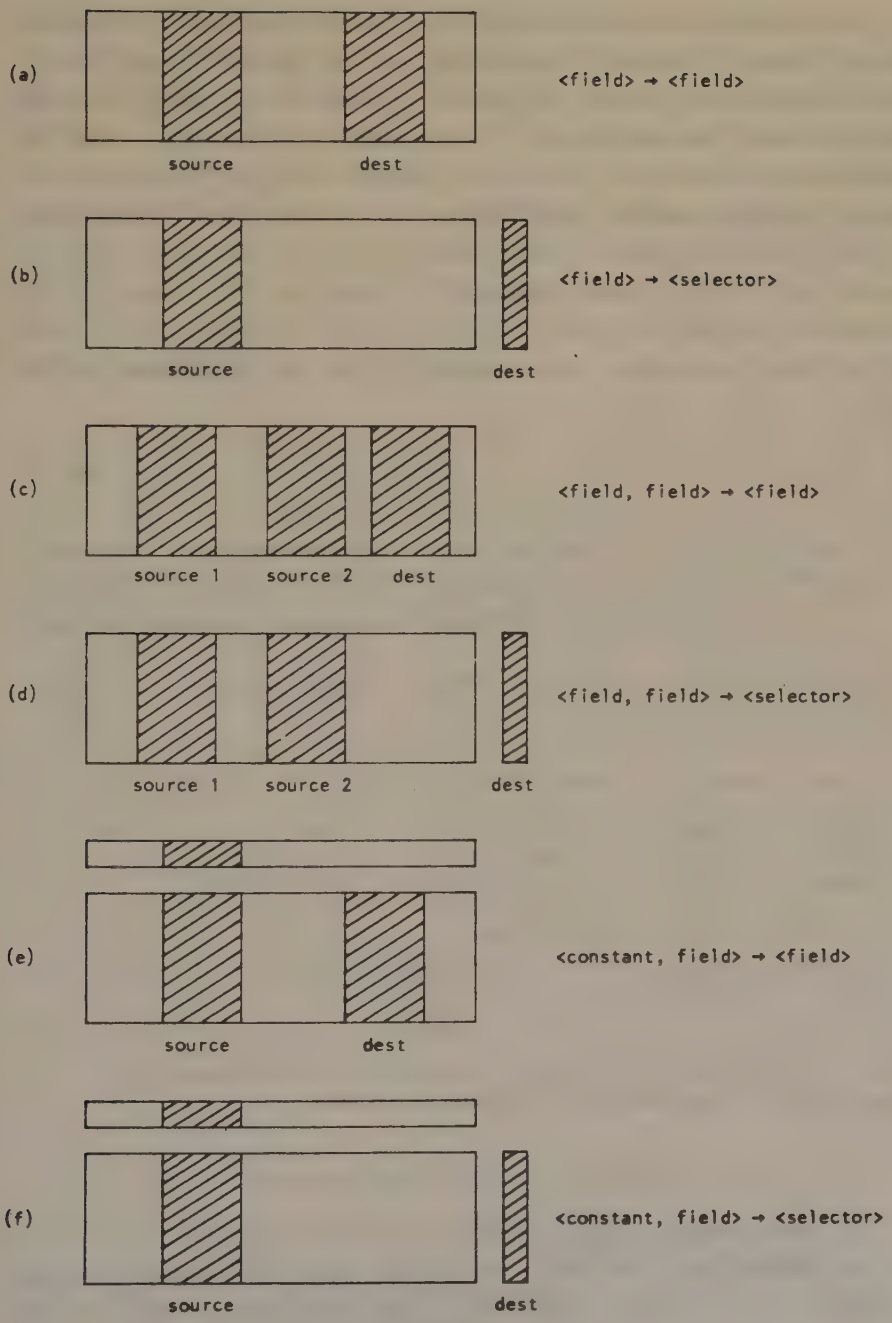


Figure 2.1.1 Basic types of instructions on the associative memory

These basic instruction types can be combined to get other types. As an example, consider the merging of two vectors to produce a third vector. To control the merge, a selector is used. In those words, where the selector is 0, the element of vector 1 will be chosen; in those words where the selector is 1 the elements of vector 2 will be chosen (see Figure 2.1.2). If we use other inputs to the Data Selector than the direct input, we can obtain many different, useful merges. The merge operation actually is the result of two successive <field>-to-<field> operations, using different source fields and different selectors, one selector being the complement of the other.

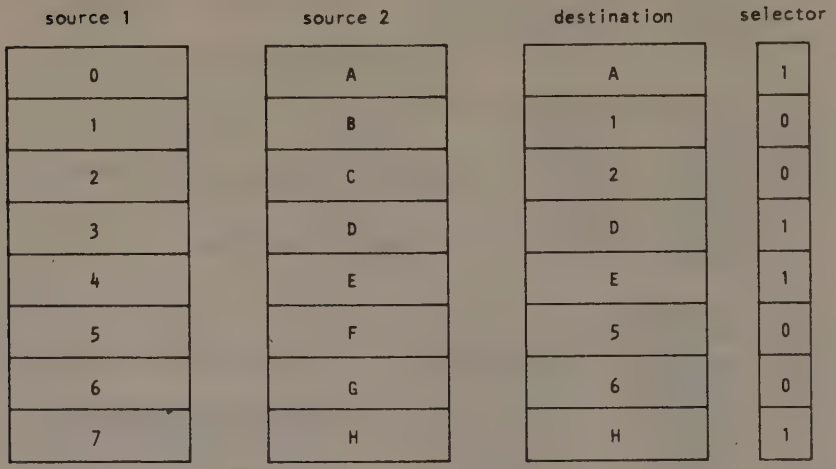


Figure 2.1.2 Illustration of a Merge operation

The specification of an instruction includes the operation and up to four parameters. The basic instruction types listed above require between two and four parameters. Types (b) and (f) require two

parameters (field address and field length). Types (a), (d) and (e) require three parameters (two field addresses and the field length). Type (c), finally, requires four parameters (three field addresses and the field length). Of course, in special cases, the value of parameters may be implied in the operation - e.g. the destination field being one of the source fields, or the length always being eight bits.

2.1.2 I/O instructions

For transfer of data between the associative memory and the outside world, and also between the associative memory and the Common and Mask Registers, the I/O registers are used. However, these can also be used for data transfer between words of the associative memory. In the latter case the I/O Buffer Register is used as temporary storage, as described in Section 1.3.2.

Instructions that involve the I/O registers are the following:

(a) <field> --> <I/O registers>

(b) <I/O registers> --> <field>
A selector may be used as a mask.

(c) <I/O registers,Selector> --> <I/O Buffer Register>

This instruction puts the contents of the selected I/O register in the I/O Buffer Register. The selector must have one single '1'.

(d) <I/O Buffer Register> --> <I/O registers>
<Common I/O register>
<Mask I/O register>

This instruction puts the contents of the source register in all I/O registers simultaneously.

I/O register instructions can likewise be combined to form instructions with other formats. As an example we consider the case of moving the value of the selected word of one field to all selected words (according to another selector) of another field. This instruction, which may be expressed as

<field,selector,selector> --> <field>

is a concatenation of the following ones:

<field> --> <I/O registers>

<I/O registers,selector> --> <I/O Buffer Register>

<I/O Buffer Register> --> <I/O registers>

<I/O Registers> --> <field> using the second selector as mask.

In the following sections (2.2 through 2.4) we will give examples of basic, widely useful, instructions. Most of the example instructions have been implemented on LUCAS and they form part of a general purpose instruction set for the machine.

2.2 MOVES, PERMUTATIONS AND MERGES

2.2.1 Introduction

In this section we present some basic instructions for moving data in the associative array. Actually, merely moving data, without changing it, is of little use in the associative array. The reason for moving data is to put it in position for operating on it. Since the communication network is between the memories and the PEs, data movement and operation can in many cases be combined. However, there are also situations where pure data movement takes place. One example is input and output to and from the array, another is the case when the data routing preceding an operation must be done in several steps because the possibilities offered by the interconnection network are limited. A context where data movements are common is data base processing. Data items are moved within the array to form new tuples in the tables in a relational data base. However, as is shown in (Kruzela, 1983) the need to move data is still relatively small, much less than when the same data structures are stored in a conventional computer.

2.2.2 Basic moves

An example of a pure move instruction is the "Load Outdata All" instruction

```
LDOA  source
```

which outputs an 8-bit field to the I/O registers. This takes one clock cycle per bit slice, plus time for parameter loading, etc, in total 11 clock cycles. The microprogram is given in the appendix, written in the microprogramming language described in the next section.

A pure move between two fields in the associative array is accomplished by the instruction

```
MOVE  source, destination, length.
```

Each bit slice takes two clock cycles to move (one read, one write). With parameter loading etc, the total time is 2b+6, where b is the field length.

The instruction

```
BROADCAST source, selector, destination, length
```

takes the contents of the source field in the first selected memory word and broadcasts it to the destination field of all words selected by the 'selector' bit-slice. First, data is moved from the source field to the I/O registers of all words, then from the selected I/O register to the I/O Buffer Register, next to all I/O registers, and finally to the destination field. 24 clock cycles are needed to broadcast an 8-bit word.

2.2.3 Use of the interconnection network

The perfect shuffle+exchange network of LUCAS is illustrated in Figure 2.2.1, using the concept of interchange boxes, introduced by Lawrie(1975).

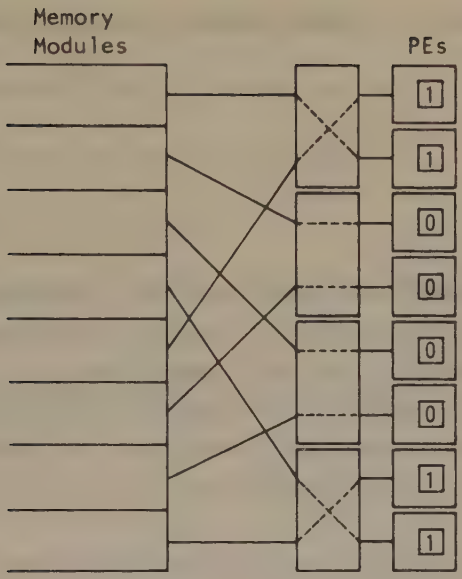


Figure 2.2.1 The principle of the perfect shuffle + exchange network in LUCAS

In reality - as described in Chapter 1 - The interchange boxes are controlled by choosing the S (as in Shuffle) or N (as in Neighbour's shuffle) input to the Data Selector of the PE. Since all Data Selectors are controlled by the same signals, the same input must be chosen for all PEs. Thus, if we want to use the straight state in some PEs and the exchange state in others, as shown in Figure 2.2.1, the transfer must be done in two steps. The state of the Tag flip-flop is used to control which input is taken. In Figure 2.2.1 the tags are '1' in the PEs that are to use the shuffle+exchange input. First, data are transferred to selected PEs using this input. Then the tags are complemented and data are transferred to selected PEs using the shuffle input. Alternatively, all PEs first receive data over the shuffle input. In PEs with T=1 these data are then overwritten with the data arriving over the shuffle+exchange input.

The latter procedure is slightly faster, since the tags need not be inverted.

Merges

Since the transfer described above is a two-step process, data may equally well be fetched from different fields in the two steps. In that case the destination field will contain data from both the source fields, intermixed according to the pattern of '1's and '0's in the tags. As an example, Figure 2.2.2 shows how the upper halves of two fields are merged to form a new field.

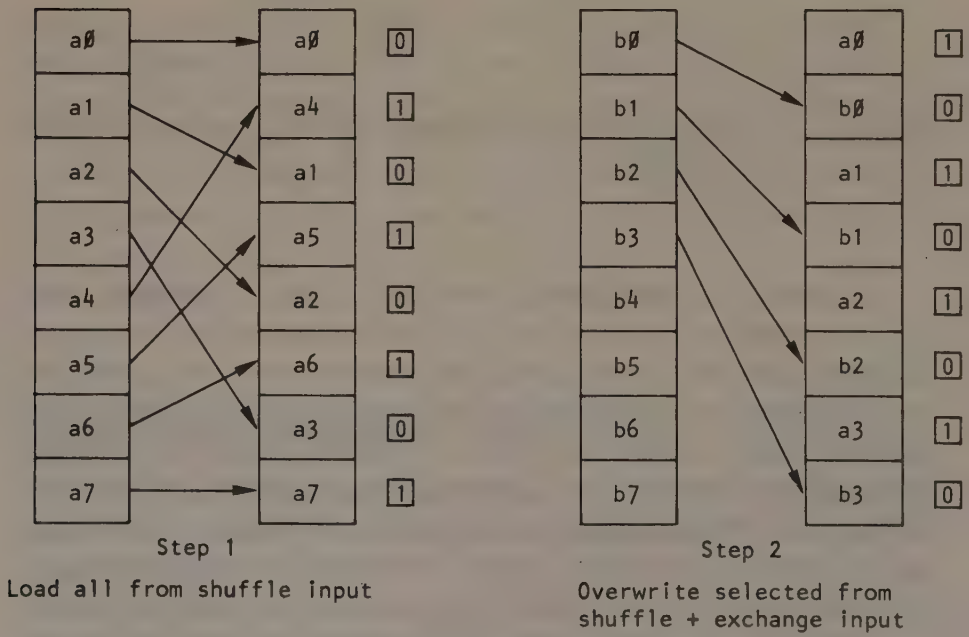


Figure 2.2.2 Merging the upper halves of two fields

In a merge operation any of the eight inputs to the Data Selector

can of course be used. A merge operation is fully characterized by

* the addresses of the two source fields

* the address of the destination field

* the length of the fields

* the Data Selector input used for data from source field 1 in PEs with T=1

* the Data Selector input used for data from source field 2 in PEs with T=0

* the selector (the contents of the tags).

All these cannot be specified in the instruction. The Data Selector inputs must be implicitly specified by the operation code. We also assume that the selector is loaded into the tags in advance. Our example operation using the N and S inputs, respectively, is then expressed

MERGENS source 1, source 2, dest, length.

In words, we express this: Merge (source 1 shuffled+exchanged) with (source 2 shuffled) into dest where T=1/0.

With the same selector, the operation MERGEAD will merge the even-numbered elements of the two sources into the destination field.

The execution time for merge operations on b-bit data is 3b+6. With b=8 this makes 30 clock cycles, or 6.0 microseconds using a 5 MHz clock. Another kind of merging operation is the following: The elements of two vectors are compared pairwise, and the maximum of them is put in the destination field. This merge is not controlled by the Tag contents but by the data itself.

2.2.4 Automatic routing

As has been shown by Lawrie(1975) there are quite a lot of frequently useful permutations that can be implemented by exactly n passes through a single-stage perfect shuffle+exchange network connecting 2^n memories with 2^n processors. Lawrie also gives a very simple algorithm to find the switch settings necessary for performing a certain permutation.

The switch settings used to guide a data item sent from source $S=\langle s_{n-1},s_{n-2},\ldots,s_1,s_0\rangle$ to destination $D=\langle d_{n-1},d_{n-2},\ldots,d_1,d_0\rangle$ are dictated by the destination only, in the following way:

The first stage brings the data item sent from S to PE no. $\langle s_{n-2},s_{n-3},\ldots,s_1,s_0,s_{n-1}\rangle$ or $\langle s_{n-2},s_{n-3},\ldots,s_1,s_0,s'_{n-1}\rangle$, depending on whether shuffle or shuffle+exchange is used. Lawrie's algorithm brings the data item to PE no. $\langle s_{n-2},s_{n-3},\ldots,s_1,s_0,d_{n-1}\rangle$, i.e. if s_{n-1} and d_{n-1} are equal, a shuffle is made, otherwise a shuffle+exchange is made. An equivalent way to state this is: if $d_{n-1}=0$ the data item is brought to the upper output of the interchange box, if $d_{n-1}=1$ it is brought to the lower output.

At the next stage, d_{n-2} determines which output is taken, i.e. now the data item arrives at $\langle s_{n-3},s_{n-4},\ldots,s_0,d_{n-1},d_{n-2}\rangle$. Thus, after n stages the destination $\langle d_{n-1},d_{n-2},\ldots,d_1,d_0\rangle$ is finally reached.

When choosing the switch setting of a certain interchange box according to this scheme, conflicts may of course arise when many data items are routed in the network simultaneously. Both inputs may, for example, want to use the lower output. Figure 2.2.3 shows a conflict-free permutation; Figure 2.2.4 shows one which cannot be implemented in n stages. In the latter figure, the unique path bringing a data item from source 001 to destination 010 has one connection - the last one - in common with the unique path from 011 to 011. Therefore a conflict occurs.

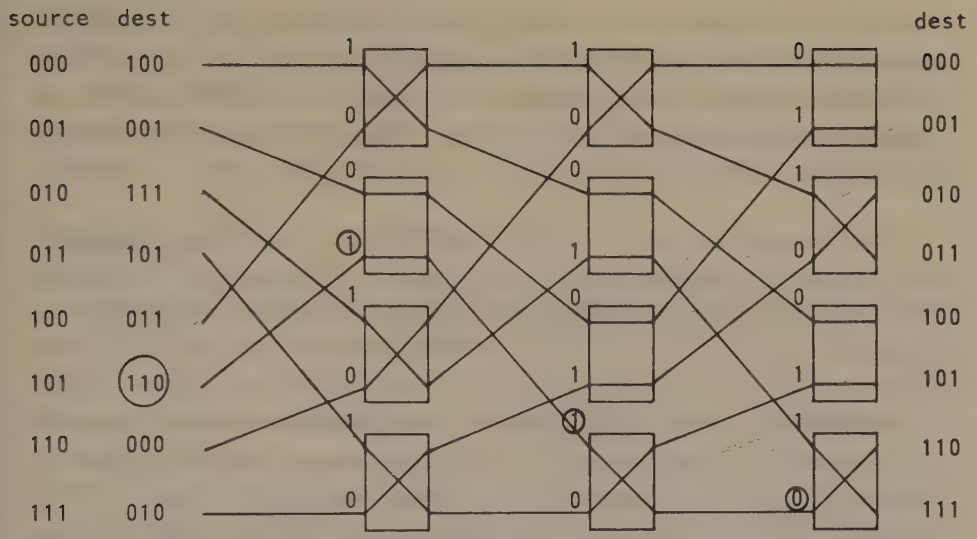


Figure 2.2.3 A conflict-free permutation. The routing of source 101 to destination 110 is specifically indicated

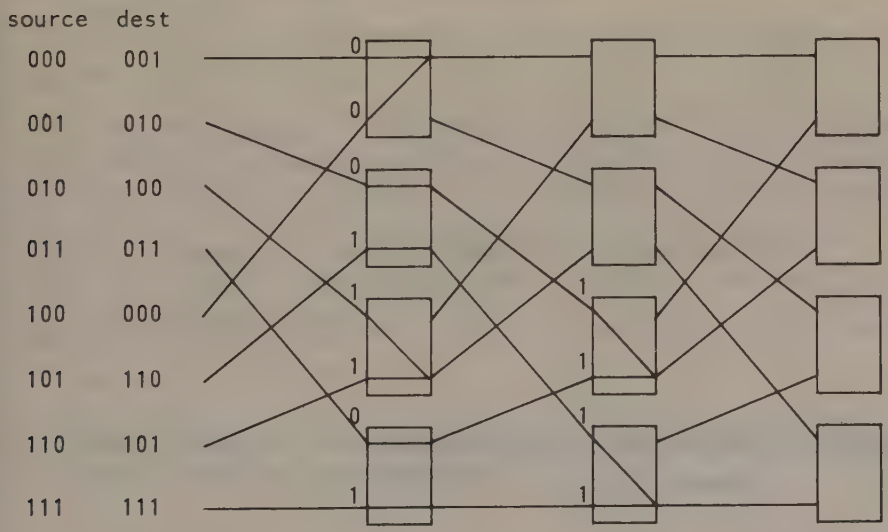


Figure 2.2.4 A permutation of 2^3 elements that cannot be performed in 3 passes

Many of the permutations that cannot be performed in n passes can be performed in 2n passes - maybe all can. Yew and Lawrie (1981) have suggested an extension of Lawrie's algorithm, which is capable of deciding switch settings for a large class of permutations that can be performed in 2n passes. The algorithm works as follows:

When a conflict occurs in the application of Lawrie's algorithm, the control bit of the upper input is given the privilege to decide. This is the same as only using the control bit of the upper input in all situations.

Unfortunately, Yew's and Lawrie's algorithm does not work for all permutations that can be carried out in 2n passes. The permutation in Figure 2.2.4 can be realized in 2n passes, as shown in Figure 2.2.5, but Yew's and Lawrie's algorithm does not find the necessary switch settings. However, used repeatedly, it finds a switch setting for 4n passes.

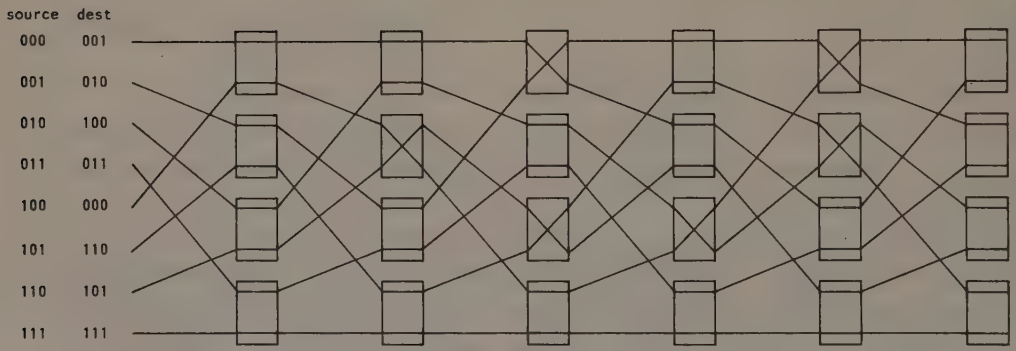


Figure 2.2.5 Realization of the permutation of Figure 2.2.4 in 6 passes

The strategy used to find the switch settings in Figure 2.2.5 can

be expressed as follows: When a conflict occurs, let the data item that has earlier been routed correctly be transferred on correctly; if both have been correctly routed earlier, or both wrongly routed, choose the "straight" state.

Implementation

Algorithms of the above kind can be used to automatically route data items to the desired destinations. The destination address is simply sent along with the data and is used to set the routing switches. The algorithm of Yew and Lawrie has been implemented on LUCAS in the following way:

Each word of the memory is assumed to contain one item of a data field and one item of a control field. The latter contains the binary address of the destination word. Furthermore, each word is assumed to have a field containing the address of the word.

Load the Tag flip-flops with the pattern
1 0 1 0 1 0 . . . 1 0.

For i = n-1 to 0:

Merge (position i of control field shuffled) with (position i of control field shuffled+exchanged) where T=1/0. Put the result in the Tags. (Now, pairs of PEs have the same Tag contents).

Merge (data+control field shuffled+exchanged) with (data+control field shuffled) where T=1/0. Put the result back in the same fields.

Next i.

Compare control field and address field. If any mismatch is found, repeat the above procedure.

The host computer can determine the success of the routing by reading the 'SOME' bit of the Status Register.

The instruction needs four parameters. The format is the following:

```
ROUTE  data, control, address, data length
```

2.3 SEARCHES, COMPARISONS AND LOGICAL INSTRUCTIONS

Search operations on a set of data elements look for elements with specific properties, e.g. the element(s) with the largest numerical value, those exactly matching the search argument, or pairs that consist of identical elements. These examples represent three different instruction types according to the classification given in part 2.1.1, namely types (b), (f), and (d), respectively.

2.3.1 Type <field> --> <selector>

The instruction "Maximum of field, tagmasked"

```
MAXT  field address, length
```

marks in the Tag registers the word(s) (among the selected ones) with largest value in the field. It proceeds from most significant bit to least significant bit discarding candidates with a '0' in the current position if there are candidates with a '1'. The microprogram is included in the appendix. The time is 3b+5 clock cycles. Thus, the maximum element in a vector of 128 8-bit data items is found in (3*8 + 5)*200 ns = 5.8 microseconds, using a 5 MHz clock.

2.3.2 Type <field,field> --> <selector>

The instruction "Compare fields, tagmasked"

```
COMPFT  source1, source2, length
```

marks in the Tag registers those words - of the selected ones - where the contents of the two fields are identical. This is accomplished by a series of successive LRMA (load R from M-input, all) and CRMT (Compare R to M-input, tagmasked) instructions to the ALU. The total time is 2b+6 clock cycles. With b=8, 22 cycles are needed, which is 4.4 microseconds, using a 5 MHz clock.

The instruction "Field greater than field, tagmasked"

```
GREAFT  source1, source2, length
```

marks in the Tag registers those words - of the selected ones - where the contents of field 1 is greater than the contents of field 2. Actually a subtraction is made, but of the result only the sign bit is used. If overflow occurs in the subtraction, the sign is inverted so that a correct result is obtained also in this case. Instruction time is 2b+7 clock cycles, which for b=8 yields 23 clock cycles, or 4.6 microseconds, using a 5 MHz clock.

2.3.3 Type <constant,field> --> <selector>

The instruction "Compare field to Common, tagmasked"

```
COCOT  field_address, length
```

marks in the Tag registers those words - of the selected ones - where the contents of the examined field is identical to the contents of the same field in the Common Register. Since bits from both operands can be input to the PEs simultaneously, this comparison only takes one clock cycle per bit. In total, b+4 clock cycles are needed. Thus, a search for an 8-bit data item in a vector of 128 items is accomplished in 2.4 microseconds.

The instruction "Greater than Common, tagmasked"

```
GREACOT  field_address, length
```

marks in the Tag registers those words - of the selected ones - where the contents of a certain field is greater than the contents of the same field in the Common Register. The ACIMA function of the ALU (see table 1.3.1) is used to subtract the bit from the Common Register from the bit from Memory, one bit each clock cycle and starting with a '1' in the Carry flip-flop. The total time is b+4 clock cycles also in this case.

We consider two versions of closest match: minimum arithmetic distance and minimum Hamming distance.

The instruction "Closest match, arithmetically, to Common, tagmasked"

```
CMACOT  field_address, length
```

computes the difference between the Common Register value and the field value in each word, then forms the absolute values of these differences, and finally searches for the minimum. The time is 7b+constant.

The search for minimum Hamming distance,

```
CMHCOT  field_address, length, counter_length
```

proceeds by first forming the exclusive OR between the Common Register value and the field value, then counting the number of ones in these results, and finally searching for the minimum value among the counts. The first phase takes 2b+constant time, the second takes 2c+constant, where c is the counter length. (A counter field is used in each word. The time can be reduced because, initially, the full counter length is not needed). The third phase is the search for minimum, which takes 3c+5 clock cycles. In total 2b+5c+constant time is used for the operation.

2.3.4 A more complex search

As an example of a more complex search instruction, we consider the following:

The maximum value in a matrix stored one row per memory word is formed by the instruction

```
MAXMAT  matrix start address, no. of columns, data length
```

The execution of this instruction proceeds in the following way:

Find the maximum element of the first column.

Move this value to that field of the Common Register that is above the second column.

Search for a larger element in this column.

If there is one, move its value to that field of the Common Register that is above the third column, else move the old value to this place.

etc.

At the end of this procedure, a field of the Common Register contains the maximum value. A bit slice in the associative array and a register in the Address Processor are constantly updated to keep track of where the maximum value so far is to be found.

2.4 ARITHMETIC INSTRUCTIONS

In this section we describe how basic instructions for addition, subtraction and multiplication are implemented on LUCAS.

2.4.1 Addition and subtraction

The instruction "Add fields, tagmasked"

```
ADDFT  source1, source2, destination, length
```

adds the contents of one source field to another, writing the result in the destination field. The implementation is a repetition of the following three ALU instructions (see table 1.3.1), constantly incrementing the address pointers.

LRMA(source1)
ADMA(source2)
WRRT(destination)

Putting the length in the loop counter of the Address Processor makes it possible to perform the loop counting and test without overhead. The total time for the instruction is 3b+8. This means that 128 pairs of 8-bit numbers are added in 6.4 microseconds, using a 5 MHz clock. The microprogram can be studied in the appendix.

The implementation of the subtract-fields instruction

SUBFT source1, source2, dest, length,

which subtracts source field 2 from source field 1, differs from the above only in that, instead of ADMA, the ADMIA function of the ALU is used, and that the carry flip-flop is set before starting. This realizes a subtraction according to the two's complement method.

Addition of a constant to a field, with the constant stored in the Common Register, is done with the instruction "Add Common to field, tagmasked"

ADCOFT source, destination, length

This takes shorter time, because bits from both arguments can be input to the ALU simultaneously. The ALU function used is ACMA, and the execution time is 2b+6 cycles.

Subtraction of a constant or subtraction from a constant are made in a similar way, using the ALU functions ACIMA and ACMIA, respectively.

2.4.2 Multiplication

Multiplication of two fields

Multiplication of two fields of two's complement numbers

```
MULFA  source1, source2, destination, length
```

is implemented using Robertson's algorithm in the following way. (The microprogram is given in the appendix).

A product field of twice the length of the source fields, minus one, is used. The bits of the multiplier in the source2 field are scanned from right to left and put in the Tags. For each bit, the multiplicand in the source1 field is added to the proper positions of the product field, but only in those words where the Tag = 1. The sign bit in two's complement numbers actually has weight = -1. Therefore, the final step is a subtraction in words with the sign bit equal to one.

The execution time is the following (b is the length of the operands):

Step 1: Loading the multiplicand, adjusted to the right, into the product field, but only in those words where the rightmost bit of the multiplier is '1'. Double sign bit is used.
Time: 3b+3 cycles.

Step 2: Clearing of the rest of the product field.
Time: b-2 cycles.

Step 3: Repeated conditional addition of the multiplicand to the product field.
Time: $(b-2)(3(b+1)+8) = 3b^2+5b-22$ cycles.

Step 4: Conditional subtraction based on the sign bit of the

multiplier.

Time: 3b+12 cycles.

Total time: $3b^2+12b-9$ cycles.

As an example, b=8 gives 279 cycles, or 55.8 microseconds.

Multiplication of a field by a constant

Multiplication of a field by a constant has a shorter execution time. The reason is that the number of additions can be reduced; only when the multiplier contains a '1' at the current position, an addition has to be made.

When multiplying by a constant, the constant may be put either in the Mask Register or in a Memory Module of the Processor Array. Both can be scanned by the Control Unit and the bits tested for zero or non-zero value. When a bit of a constant stored in the array is to be tested, a selector is used to point out the word, the bit is loaded to the Tag register, and the SOME-line is interrogated.

The average time for multiplication by scalar will be half the time for multiplication of fields.

A further reduction of the number of add-type operations can be achieved if the multiplier is recoded into a form that has fewer non-zero bits. The signed-digit (SD) code is such a form (Hwang, 1979). In an SD number, the allowed digit set is {1,0,1}, where 1 stands for -1. The weights of digits are the usual powers of two.

The SD code of a number is not unique. For example, the decimal number 7 can be coded into a 5-digit SD vector in three different ways: 00111, 01001, and 11001. One of these has a minimal number of non-zero digits, but in general not even the minimal form is unique.

A minimal SD vector that contains no adjacent non-zero digits is

called a canonical signed-digit (CSD) vector. Such a vector can be obtained by the following procedure (Figure 2.4.1):

Starting at the least significant bit, replace sequences of m '1's, where m>1, by m-1 '0's preceded by a '1' and succeeded by a '1'.

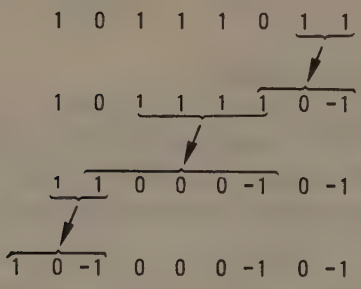


Figure 2.4.1 Recoding a number to canonical signed digit code

A microprogram to convert a vector stored in a field of the associative memory to CSD form can be found in the following way:

The procedure described above is realized by an automaton scanning the bits of B from right to left and with the state transition graph shown in Figure 2.4.2.

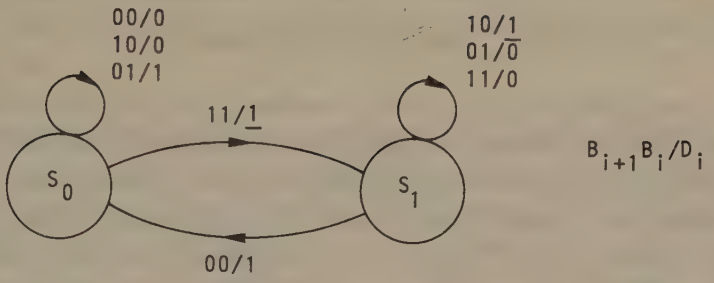


Figure 2.4.2 State transition graph for bit serial recoder

s_0 is the initial state. As long as B_i=0 we stay in s_0, outputting D_i=0. When a single '1' is found, D_i=1 is output. When B_{i+1}=B_i=1 is encountered, D_i=$\underline{1}$ is output and the state is changed. As long as B_i=1 we stay in s_1, outputting D_i=1. B_i=0, B_{i+1}=1 will cause us to output $\underline{1}$, still staying in s_1. Only if two consecutive '0's are found will we return to s_0, outputting a '1'.

Coding s_0 as q=0 and s_1 as q=1 gives the following next-state function q^+:

$$q^+ = qB_{i+1} \vee B_i$$

The output is three-valued. We choose the following code:

	u_1	u_2
0	-	0
1	0	1
$\underline{1}$	1	1

This gives the expressions

$$u_1 = q^+$$

$$u_2 = q + B_i \text{ , where + is add modulus 2}$$

If we equip the ALUs with functions that suit these expressions exactly, the conversion can be made at maximum speed, which is three cycles per bit-slice (one read and two writes). If we want to use the standard ALU function set, 6 cycles per bit-slice will be needed. The state is held in both the R and C flip-flops. The following sequence of functions implements the above functions (j and j+1 are the bits written into when bit no. i is scanned):

	Contents of flip-flops	
	C	R
	q	q
XORRMA, address i		$q+B_i$
WRRA, address j		
LRMA, address i		B_i
ADMA, address i+1	$q^+(=u_1)$	
LRCA		$q^+(=u_1)$
WRRA, address j+1, incr j, loop		

In the appendix, the routine is given in the high level microprogramming language described in Chapter 3.

The relevance of doing the conversion to CSD form in the Processor Array will be seen in Chapter 4, where matrix multiplication is described. During the execution of the algorithm,

each element of one of the matrices is to be used as a multiplier once. Therefore it is convenient to do the conversion to CSD form of an entire column simultaneously. This is probably a rule for data produced in the array, if one item is going to serve as a multiplier, many items are.

Using a CSD coded multiplier, the number of add-type operations (additions and subtractions) in a multiplication will on the average be b/3, where b is the length of the multiplier (Hwang, 1979). Therefore, the average time for multiplication by a CSD coded scalar will be one third of the time for multiplication of fields. For example, 8-bit multiplication takes 19 microseconds, on the average.

2.5 SUMMARY OF EXECUTION TIMES

Table 2.5.1 gives a summary of execution times of basic operations that we have dealt with in this chapter. To be able to compare the speed of the 128-PE LUCAS with the speed of a sequential computer we have also calculated the time in which a sequential computer is required to perform the same operation on one data item, or on a pair of data items, depending on the instruction type. We call this time the "Equivalent Sequential Time per operation", or EST. It is given in nanoseconds.

For example, if a sequential computer is to have the same performance as LUCAS, it must be able to add two 8-bit numbers in 50 ns, including data fetch and storage, addressing and loop counting.

	8-bit data		16-bit data	
	Micro-seconds	EST	Micro-seconds	EST
MOVES AND MERGES				
Load Outdata (8-bit only)	2.2	17		
Move field	4.4	34	7.6	59
Broadcast field	4.8	38	9.6	75
Merge fields	6.0	47	10.8	84
SEARCHES AND COMPARES				
Maximum of field	5.8	45	10.6	83
Compare fields	4.4	34	7.6	59
Field greater than field	4.6	36	7.8	61
Compare field to Common	2.4	19	4.0	31
Field greater than Common	2.4	19	4.0	31
ARITHMETIC				
Add (Sub) fields	6.4	50	11.2	88
Add Common to field	4.4	34	7.6	59
Multiply fields	56	436	190	1484
Multiply field by Common (aver.)	28	219	95	742
Multiply field by CSD-coded Common (average)	19	145	63	492

Table 2.5.1 Execution times for some basic instructions. EST is the Equivalent Sequential Time per operation

CHAPTER 3
SYSTEM SOFTWARE

Hardware and software have been advanced in parallel in the course of the project. In this chapter we will give an overview of the system software developed for LUCAS. It consists of (1) a debugging monitor, useful for microprogram testing and hardware debugging, (2) a microprogram assembler, (3) a microprogram compiler and (4) a high level language specification. More detailed material can be found in (Fernstrom, 1983) and (Kruzela, 1980).

3.1 DEBUGGING MONITOR

A debugging monitor, AMON, has been implemented on LUCAS, mainly for the purpose of debugging microprograms. It has also been an unvaluable tool for debugging the hardware. AMON allows microprograms to run at full speed on LUCAS with a breakpoint facility. The contents of both the Microprogram Memory and the Associative Memory may be displayed, changed, loaded from or written to disk.

The commands in AMON are divided into three groups:

* Microprogram Memory commands allow the user to load microprograms from disk, display and change the contents of the Microprogram Memory, and insert breakpoints in the microprograms.

* Execution commands allow the user to load the Parameter

Registers and the Instruction Register, thereby starting the execution of a microprogram.

* Associative Array commands allow the user to display the contents of the Associative Memory using different formats (Hexadecimal, ASCII characters, binary, 8 or 16 fixed point numbers) and to change the contents. Also the contents of the PE flip-flops and the Common and Mask Registers can be displayed and altered. Furthermore, an arbitrary area of the Associative Memory can be stored on disk or loaded from disk. It can also be sent to the image display system using an arbitrary pixel size.

3.2 ASSEMBLY LANGUAGE FOR MICROPROGRAMMING

A microprogramming language of assembly type has been defined for LUCAS. A microprogram assembler is implemented in the form of a macro definition library used with the macro assembler of the Host computer.

Here we give the information necessary for understanding the microprogram examples given in this thesis.

An instruction in the language is divided into maximum four fields: AM, DEF, LABEL and DATA. The field designation must preceed the mnemonics.

AM field

The AM field specifies the instruction for the Processor Array. It consists of

* ALU function, i.e. one of the mnemonics listed in table 1.3.1.

* Data Select code, which selects one of eight inputs to the Data Selector: D (direct), S (shuffle), N (neighbour's shuffle), A (above), B (below), E (east), W (west), X (X flip-flop). As shown in Figure 1.3.14, E and W are currently not implemented. The default value of the code is X.

* Memory Write Control: WRRT and WRRA (Write data from the R flip-flop in selected words, and in all words, respectively), WRIT and WRIA (Write data from the I/O register in selected words, and in all words, respectively). The valid memory address is the one specified in the preceding microinstruction.

* I/O Register Control code: SHIFT, IORS (Read selected I/O register), IOWRA (Write in all I/O registers).

* Select First control: SELF

* Network control: NETEN (Network Enable), which causes one bit from each Memory Module to be output on the 128-bit data bus. It must be used when other inputs to the Data Selector than D or X are used.

DEF field

The DEF field specifies the instruction for the Control Unit. It consists of

* Microprogram Sequencer Instruction Code. See table 1.3.2.

* Condition Code. Selects test condition for the Sequencer. Available conditions are listed in Section 1.3.4. The status after the preceding microinstruction is tested.

* Address Processor Function. The 16 functions are listed

in Section 1.3.4.

* Address Processor Input Source. Selects input to the Address Processor: PARAM1..4 (The contents of the Parameter Register 1..4), IOBUFF (The contents of the I/O Buffer Register), PPL (The contents of a 12-bit field in the Pipeline Register holding the microinstruction).

* Register Select Code. Selects registers in the Address Processor through an A-address RA1..RAF, and a B-address RB1..RBF. See Section 1.3.4.

* Stack Control: SPOP (Pop the Address Processor stack), SPUSH (Push the Address Processor stack).

* Loop Counter Control: DECLC (Decrement Loop Counter), LDLC (Load Loop Counter).

* Common and Mask Write Control: WRC (Write Common), WRM (Write Mask), WRCM (Write Common and Mask).

* I/O Buffer Control: ENIO (Enable I/O Buffer Register output), LDIO (Load I/O Buffer Register).

* Host Interface Control: NEXT (Reset the BUSY flip-flop), LDSTAT (Load the Status Register with the status information).

LABEL field

The LABEL field has one command, SET, and a parameter which is used as input to the Sequencer (jump address or Register/Counter value).

DATA field

The DATA field has one command, SET, and a parameter, which is used as data for the Address Processor.

Example

As an example of a microinstruction,

```
DATA    SET OFFOH
LABEL   SET WRICOM
AM      COT,WRRA
DEF     CJS,SOME,LDBD,RBD,PPL,LDSTAT,NEXT
```

complements the Tag flip-flops, writes the contents of the R-flip-flops into the memory of all words at the bit-address specified in the preceding microinstruction, furthermore loads register D of the Address Processor with the value FF0, loads the Status Register, resets the BUSY flip-flop and makes a subroutine call to label WRICOM if some Tags are '1' after the previous microinstruction.

Examples of microprograms in the assembly language are given in the appendix.

3.3 HIGH LEVEL LANGUAGE FOR MICROPROGRAMMING

In order to simplify microprogramming, a microprogramming level that uses high-level constructs has been developed. The part of LUCAS which is the most difficult to program is the Control Unit, and especially the Address Processor will become a bottleneck if not properly handled. The use of local and global variables in the language, arithmetic expressions on variables and automatic handling of parameter passing to subroutines make the Address Processor disappear from the programmer's view.

By using high-level like control constructs, no explicit programming of the Sequencer need to be done and the readability of the programs is greatly improved.

What remains are the instruction for the Processor Array, the Common, Mask, I/O and I/O Buffer registers. The same mnemonics are used as in the assembly language described above.

The language is described in detail in (Fernstrom, 1983) together with a description of the compiler and the techniques used to optimize the code. As efficient code can be generated using this language as using the assembly language.

We give example of microprograms in Chapter 4 and in the appendix.

3.4 HIGH LEVEL LANGUAGE PASCAL/L

A high level language which takes advantage of the machine architecture has been specified (Fernstrom, 1982, 1983). The language is an extension to Pascal named Pascal/L.

The extensions needed to describe parallel processing are of two kinds: new data types and means to handle them, and new constructs in the control structure.

New data types and expressions

A variable of the type selector is defined as a Boolean vector with length less than or equal to the number of available PEs. Selectors are used for selecting PEs for parallel processing. They serve as activity vectors held in the Tag flip-flops at execution time.

A field in the Associative Memory is represented by a variable of the type parallel array. When a parallel array is declared, the first

dimension specifies its range of parallelism. This range defines in which PEs the elements of the array are located. For example,

```
var para: parallel array [0..99,0..2] of integer(12)
```

declares the variable para to be defined in the words 0 to 99 in the Associative Memory. Each word contains three 12-bit elements (indexed 0..2) of para.

It is possible to combine sequential and parallel variables in expressions as long as no type conflict occurs. As an example, the meaning of

```
4 * para[*,1]
```

is that each element of column 1 of para is multiplied by 4. As another example,

```
para[*,0] > 4
```

selects those words where the value in column 0 is greater than 4.

Extended control structure

New elements of the control structure are motivated by the parallel execution. The if, case, while, repeat and for statements of Pascal are extended with where, case where and while and where in Pascal/L.

As an example, supposing A and B are vectors, the statement

```
while and where A>B do A:=A-B
```

will give A[i] modulus B[i] for each item of A.

No compiler for Pascal/L has been implemented. Still, the language has been useful for the purpose of expressing algorithms. Examples are given in Chapter 4.

CHAPTER 4
SOME BASIC ALGORITHMS

4.1 INTRODUCTION

It is often seen that computations performed in separate application areas rely on common mathematical tools and computational techniques. If it can be shown that a particular computer design is well suited for the application of one or several widely used tools, it means that the computer may be useful in many application areas.

In this chapter we study the implementation on LUCAS of three important classes of computations, namely matrix multiplication, computation of the discrete Fourier transform (DFT) by means of the fast Fourier transform (FFT) algorithm, and solution of graph theoretical problems. They all represent tools and techniques that are not limited to any specific realm of computation.

Studies of matrix multiplication on parallel computers have been made in connection with the DAP project (Flanders et al, 1977) and in (Pease, 1977). The DAP interconnection scheme is very different from the one used on LUCAS, which results in different solutions to the data routing problem. The approach taken by Pease on a proposed cube-connected processor array also differs widely from the methods reported here.

The FFT algorithm for the calculation of the DFT has been known since 1965 (Cooley and Tukey, 1965). It has also been well known that it can be mapped efficiently onto a perfect shuffle-connected processor array (Pease 1968, Stone 1971). However, this is probably the first time that it has been done in practice.

Graph theoretic problems are relevant in many application areas. Very often they are open to efficient solution by parallel computers. An algorithm for solving the shortest path problem on LUCAS is given. We also demonstrate on other graph theoretic problems how algorithms designed to be efficient on conventional computers can be adapted to a parallel computer like LUCAS.

The three computational areas dealt with in this chapter put rather diverse demands on LUCAS. Matrix multiplication and Fourier transformation utilize the parallel interconnection network to a high degree and require that it be effective, so that full parallelism can be maintained all the time and so that communication can be carried out without conflicts. The graph theoretic problems require that searches be performed efficiently, and also that some quite unconventional data passing be performed. LUCAS turns out to meet these diverse demands fairly well.

4.2 MATRIX MULTIPLICATION

The multiplication of two n by n element matrices consists of the formation of n^2 inner products of pairs of n-element vectors. An inner product of two vectors is defined as

$$\langle a_1,a_2,\ldots,a_n\rangle * \langle b_1,b_2,\ldots,b_n\rangle = a_1b_1 + a_2b_2 +\ldots+ a_nb_n.$$

When multiplying two matrices, A and B, the element of the i:th row of column no. j is formed as the inner product of the vectors comprising row no. i of A and column no. j of B.

The traditional method for multiplication of two matrices computes the inner products sequentially, one after the other. Each inner product is likewise computed sequentially, as a sequence of multiplications and additions.

Parallelizing the computation can be made in many various ways. Depending on the number of processors available compared to the size of the matrices, different approaches may be favourable.

4.2.1 n x n matrices, n processors

We first consider the case when n processors are available. The most obvious way of using the parallelism is to calculate the n multiplications of an inner product computation simultaneously. This is known as the "inner-product method" (Hockney and Jesshope, 1981). The remaining addition of the terms of each inner product can be made by n processors in O(log n) time using a suitable communication network, e.g the perfect shuffle+exchange network. Since, on a bit-serial computer, the execution time for addition is small compared to the multiplication time, the reduced parallelity in the addition step is not very severe. However, there are other problems with this approach:

The two vectors that we multiply in order to compute an inner-product must be aligned with each other, so that corresponding elements are available by the same processor. If the A-matrix is stored one column in each processor's memory, the B-matrix must be stored one row in each memory. This will align the rows of A with the columns of B.

If the matrices are loaded into the memory for multiplication only, this need not cause any problem. However, if the matrices are created in the parallel array or are subject to other operations that perhaps demand the same storage method for both, the alignment problem has to be solved in the array. STARAN has this possibility through its Multidimensional Access Memory (Batcher,1977), where both rows and columns are accessible. This is not possible on LUCAS.

Instead of computing each inner product with the largest possible parallelity, many inner products can be computed simultaneously. (A total of n^2 inner products are to be computed).

Referring to Figure 4.2.1, in order to form the first column of the result matrix, the following n inner products must be computed:

$$\langle A_{00}A_{01}A_{02}\cdots A_{0,n-1}\rangle * \langle B_{00}B_{10}B_{20}\cdots B_{n-1,0}\rangle$$
$$\langle A_{10}A_{11}A_{12}\cdots A_{1,n-1}\rangle * \langle B_{00}B_{10}B_{20}\cdots B_{n-1,0}\rangle$$
$$\vdots$$
$$\langle A_{n-1,0}A_{n-1,1}\cdots A_{n-1,n-1}\rangle * \langle B_{00}B_{10}B_{20}\cdots B_{n-1,0}\rangle$$

These expressions show that the first column of A is to be multiplied by B_{00}, the second column by B_{10}, the third by B_{20}, etc. The results of these multiplications are accumulated, and the final results will appear at the correct positions.

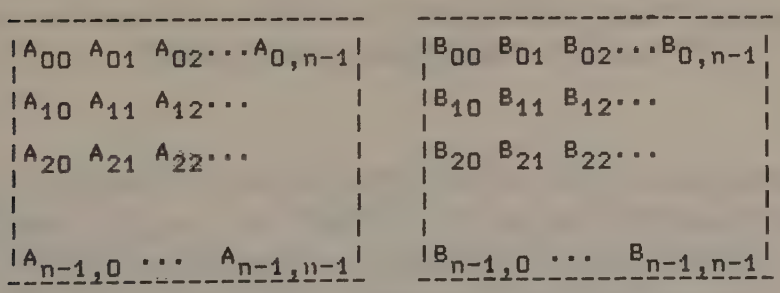


Figure 4.2.1

Thus, the k:th column of the product is formed by successively multiplying each column of A by the elements of the k:th column of B, constantly accumulating the results. In this scheme, the multiplications are made as multiplications by a scalar, which frees

us from the problem of vector alignment. Figure 4.2.2 illustrates the algorithm, called the "middle-product method".

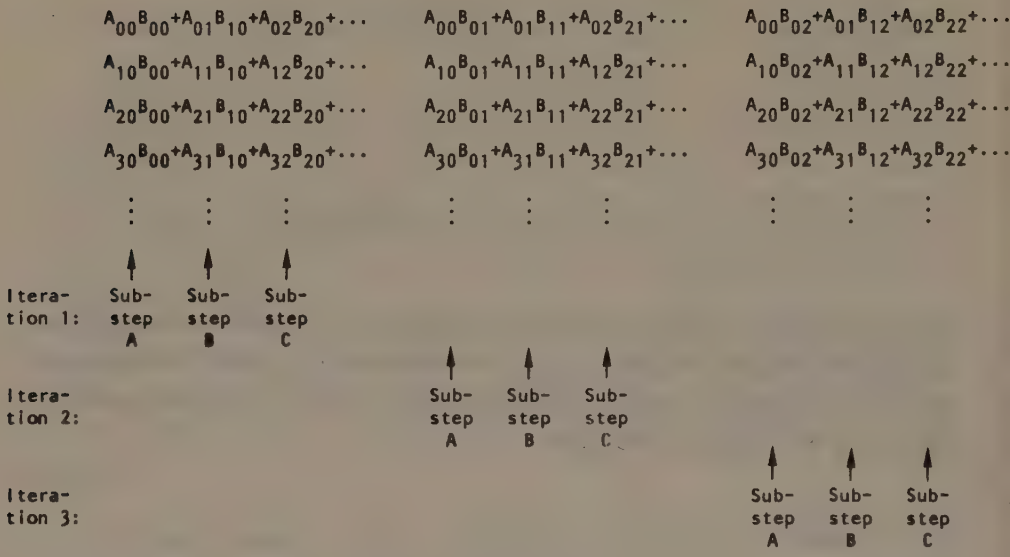


Figure 4.2.2 Matrix multiplication algorithm that produces the result column by column (middle-product method)

We arrived at the new scheme by stating that we wanted the result to come out one column at a time, then analyzing which data and which computations were needed to produce the result in this form. Continuing this line of reasoning, we may ask if we can find a method to produce the whole result matrix simultaneously, i.e. proceed in n iterations, where each iteration produces an n x n matrix, and the last one produced is the result matrix. We want all inner products to "grow" simultaneously, as illustrated in Figure 4.2.3.

As can be seen from the figure, this can be done without problems. In fact this is just doing things in different order compared to the method described above. The access problems (or, rather, lack

of problems) are exactly the same. This is what is called the "outer-product method".

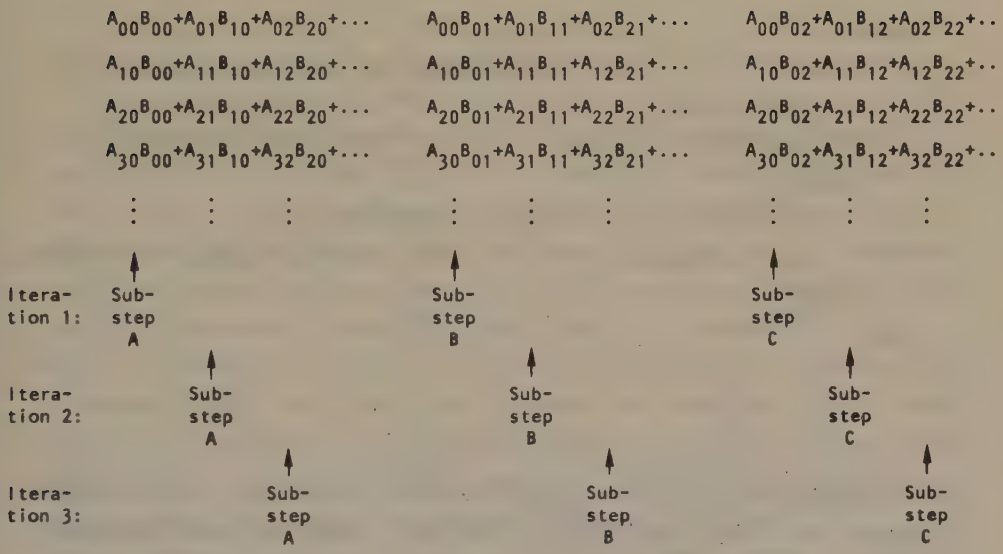


Figure 4.2.3 Matrix multiplication algorithm that computes all inner-products "simultaneously" (outer-product method)

The two methods described work equally well if the matrices are not square. The only constraints on the size are that the number of rows must be smaller than or equal to the number of processors, and that the memory space horizontally is large enough. The middle-product method which produces the result column by column is more space conservative, since the B matrix can be successively overwritten if desired.

Written in Pascal/L the middle-product method looks as follows:

```
Program MATRIXMULT;
var A,B,C: parallel array[0..127,0..127] of integer(8);
row: selector[0..127]:=(0<=True);
acol,bcol: integer;

begin
  for bcol:=0 to 127 do C[*,bcol]:=0; (* clear C *)
  for bcol:=0 to 127 do              (* for each B-column *)
    for acol:=0 to 127 do            (* multiply all columns of A *)
    begin
      C[*,bcol]:=C[*,bcol] + B[row,bcol]*A[*,acol];
      row:= rotate(row,1);           (* with each element of B-column *
    end;
end.
```

The program for the outer-product method is obtained from the above program by just interchanging the two innermost for-statements.

An estimation of the time required to multiply two 128 x 128 element matrices of b-bit data can be made as follows:

The number of multiplications by scalar are $128^2 = 2^{14}$. If recoding of the multiplier using canonical signed-digit code is done (see Chapter 2), a multiplication will - on the average - consist of b/3 additions (Hwang, 1979). Each addition takes a time close to 3b cycles. Each multiplication is preceded by a transfer of the multiplier to the Mask Register, which takes approximately 4b cycles. It is followed by an addition over 2b bits. Thus, the multiplication time is approximately

$$2^{14} * (b/3 * 3b + 4b + 6b) = 2^{14}(b^2+10b) \text{ cycles.}$$

For b=8 this is approximately $2.4*10^6$ cycles. With 200 ns clock

cycles the time is approximately 0.5 seconds, overhead time not included. The time for recoding is negligible.

4.2.2 n x n matrices, n^2 processors

We next consider the case when we have n^2 processors available for the multiplication of two n x n matrices. Since there are n^2 inner products to be computed independently of each other, the n^2 parallelism can always be fully utilized. However, data alignment before multiplication may cause some overhead. We will present two alternative methods of doing the computation on a perfect shuffle-connected processor array. The first method is a computation in place method, i.e. each inner product is computed in its final place. This requires a significant amount of shuffling before each multiplication. The other method computes the different terms of an inner product in different processing elements. The final rearrangement of the result is done in the summation phase.

Outer-product method

This method computes all inner products simultaneously in a total of n iterations. The method is a further parallelization of the one illustrated in Figure 4.2.3. All substeps in one iteration are done in parallel. This requires the data alignment depicted in Figure 4.2.4. In iteration k, a matrix $A_c(k)$ is formed by putting n copies of column k of A alongside of each other. This matrix is multiplied, element by element, with the matrix $B_r(k)$, formed as n copies of row k of B. The n^2 products form the k:th term of all the inner products, all placed at the proper positions in the matrix. All such I(k) matrices are successively accumulated.

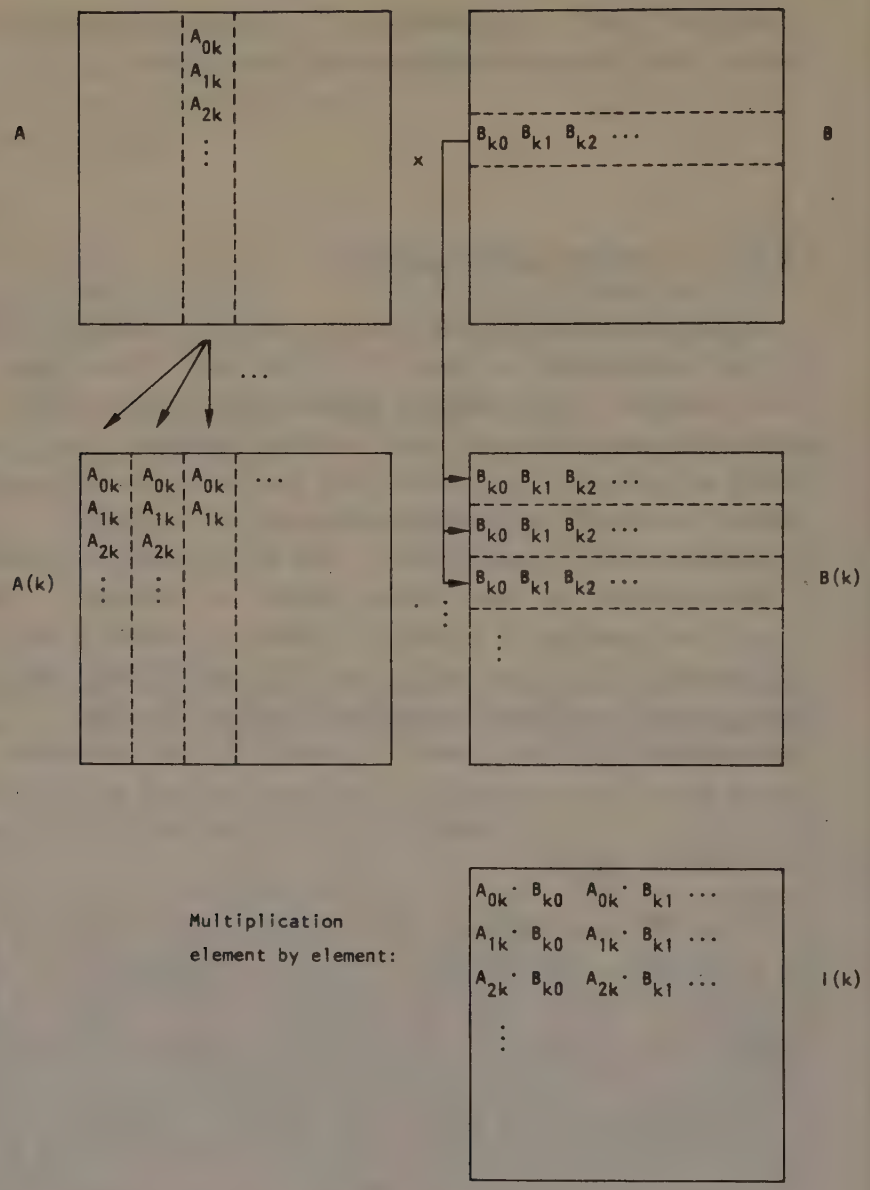


Figure 4.2.4 Alignment needed for using n^2 parallelism according to the outer product method

On a parallel processor where the processing elements are connected in a square array, like DAP, this computation is easily accomplished. The distribution of a column or a row to all other columns and rows, respectively, can be done via horizontal and vertical "highways" (Reddaway,1979). Full n^2 parallelism is used during the whole computation. The method works equally well for other matrix sizes, m x p, as long as both m and p are smaller than n, the size of the side of the array.

On a shuffle-connected array, the n-times duplications of columns and rows cannot be made as efficiently as in a square-connected array with highways. In one pass through a single-stage interconnection network, data can at most be duplicated, using the broadcast states of the interchange boxes. Thus, the formation of $A_c(k)$ and $B_r(k)$ take at least $\log_2 n$ passes each.

If we assume that matrices A and B are stored columnwise, the formation of $A_c(k)$ and $B_r(k)$ becomes as shown in Figure 4.2.5.

Both patterns can be achieved by multiple passes through the shuffle-exchange network. Broadcasting a column of the matrix through $\log_2 n$ stages, as shown in Figure 4.2.6 (for n=4) gives a result which is the transpose of the desired A_c matrix. As shown by Stone(1971), an n x n matrix can be transposed in $\log_2 n$ perfect shuffles. Thus, A_c can be derived from A in $2*\log_2 n$ passes through the network. In order to create B_r from B, B is first transposed. Then the desired column is broadcast in $\log_2 n$ passes. Figures 4.2.7 and 4.2.8 show the formation of $A_c(1)$ and $B_r(1)$ for the case n=4.

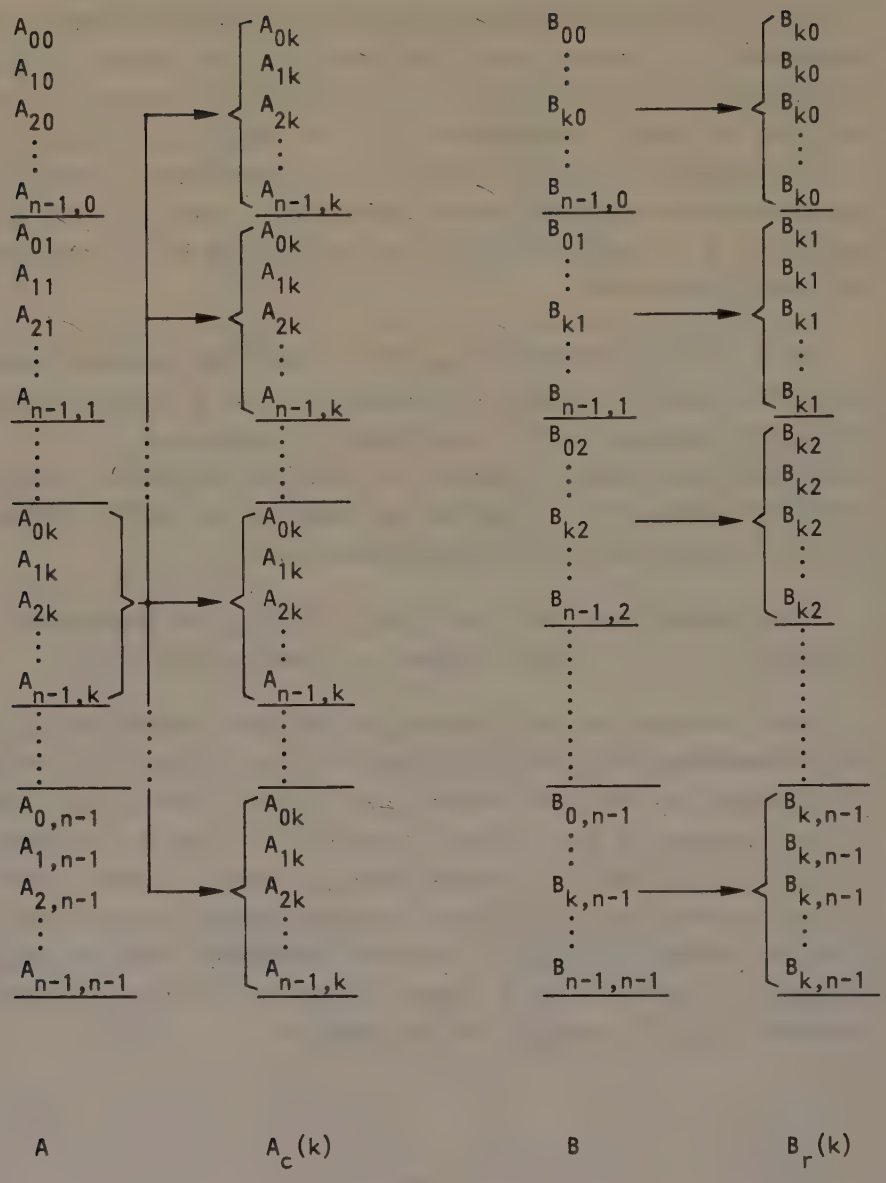


Figure 4.2.5 Broadcasting a column of A and a row of B in order to align data for iteration no. k

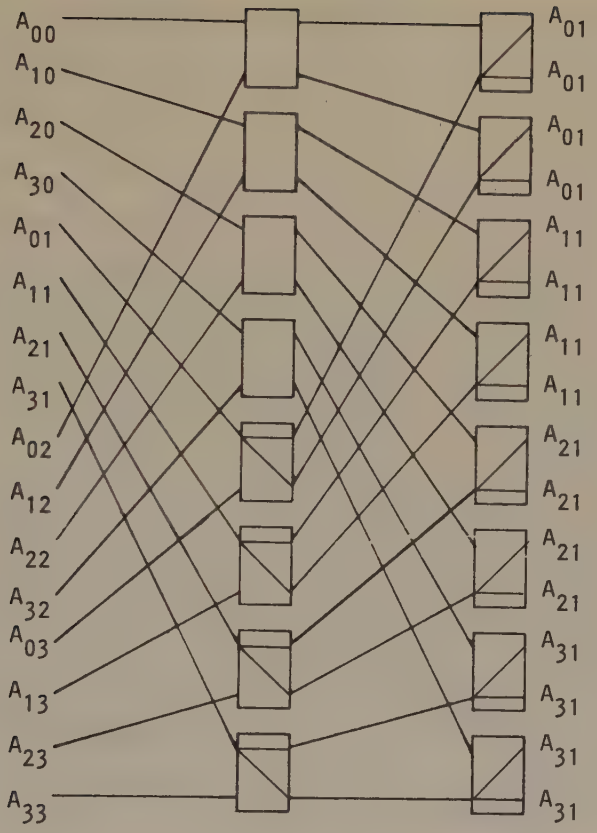


Figure 4.2.6 Broadcasting a column through $\log_2 n$ stages (n=4)

As can be seen in Figure 4.2.6, broadcasting of different columns can partly be done simultaneously. When column 1 is broadcast to the lower half of the memory, column 0 can simultaneously be broadcast to the upper half. With n columns to be broadcast, n/2 columns can be treated simultaneously in the first broadcast step, n/4 in the next step, etc. The total number of broadcasts will be

$$2 + 4 + 8 + \ldots + n = 2(n-1).$$

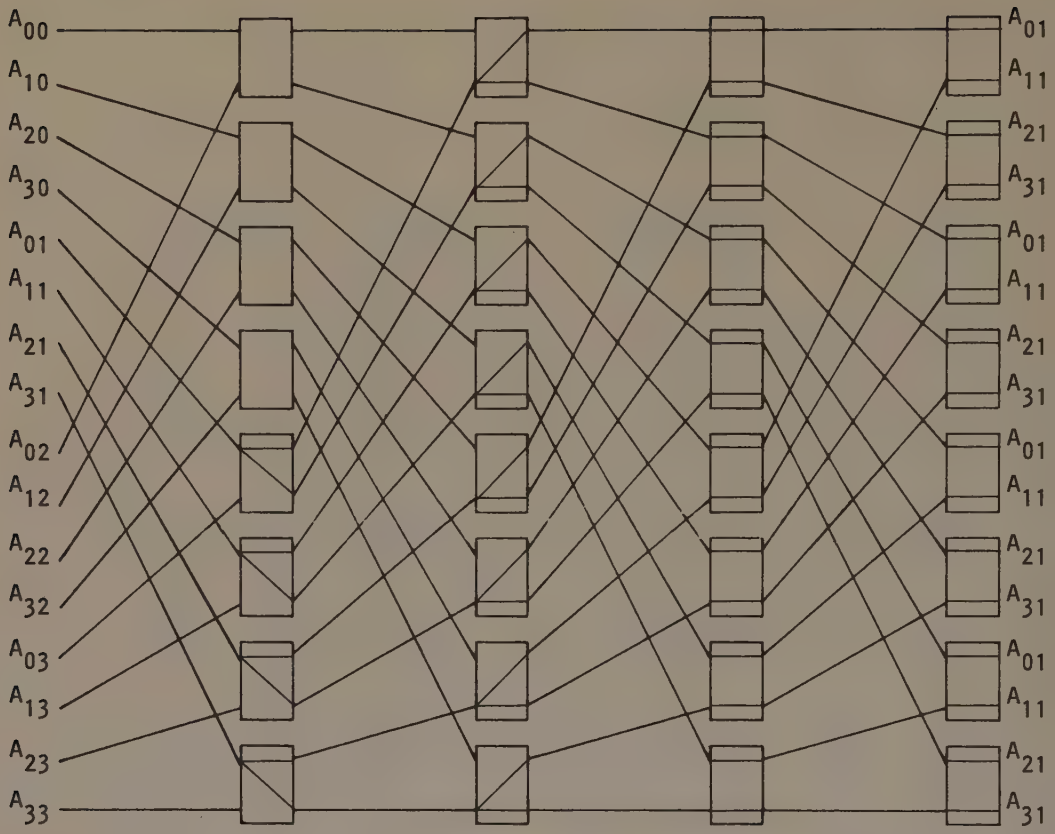


Figure 4.2.7 Formation of $A_c(1)$ from A

The formation of all A_c matrices will require these 2(n-1) broadcasts followed by n transpositions, each requiring $\log_2 n$ shuffles. On LUCAS, passes through the network that use the broadcast state consist of two reads and one write each, compared to one read and one write in an ordinary pass. Thus they take 50% longer time. Consequently, the 2(n-1) broadcasts are equivalent in time to 3(n-1) passes through the network. The total number of passes, then, needed to form all A_c matrices is

$$3(n-1) + n\log_2 n.$$

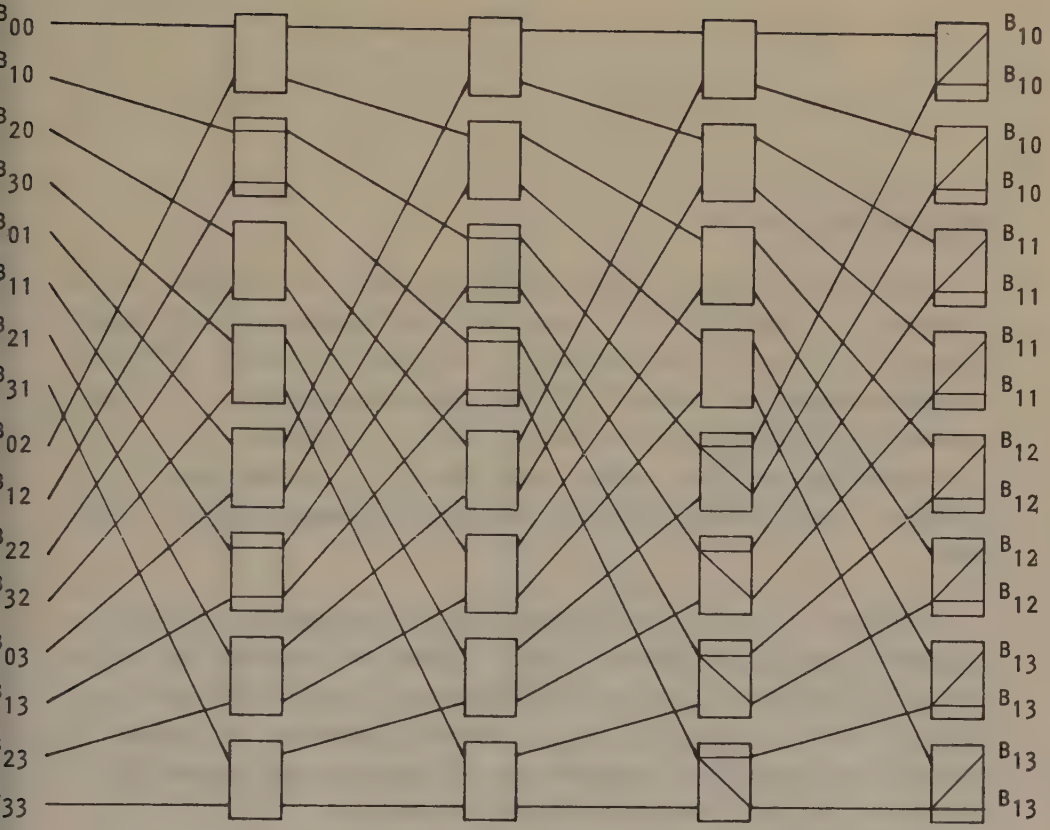


Figure 4.2.8 Formation of $B_r(1)$ from B

The number of passes to form all B_r matrices is

$$\log_2 n + n\log_2 n = (n+1)\log_2 n.$$

Thus, the total number of passes through the shuffle-exchange network on LUCAS in order to align arguments for multiplication on the final places is

$$P_1 = 3(n-1) + (2n+1)\log_2 n.$$

Middle-product method

The above method was the result of doing all substeps of an iteration in Figure 4.2.3 in parallel. This resulted in all inner products "growing" simultaneously in a total of n steps. With the method described in Figure 4.2.2, the entire computation of one inner product is finished before the next is started. A further parallelization of this method will finish the computation of n different inner products before starting the computation of n new ones.

In order to perform all substeps of an iteration in parallel, the alignment depicted in Figure 4.2.9 is needed. In iteration no. k the matrix $B_c^T(k)$ with n identical rows is formed. Each row is column no. k of B. This can be done in $\log_2 n$ broadcast-shuffles, as shown in Figure 4.2.10. With this alignment, the n^2 multiplications needed for column no. k of the result matrix can be computed.

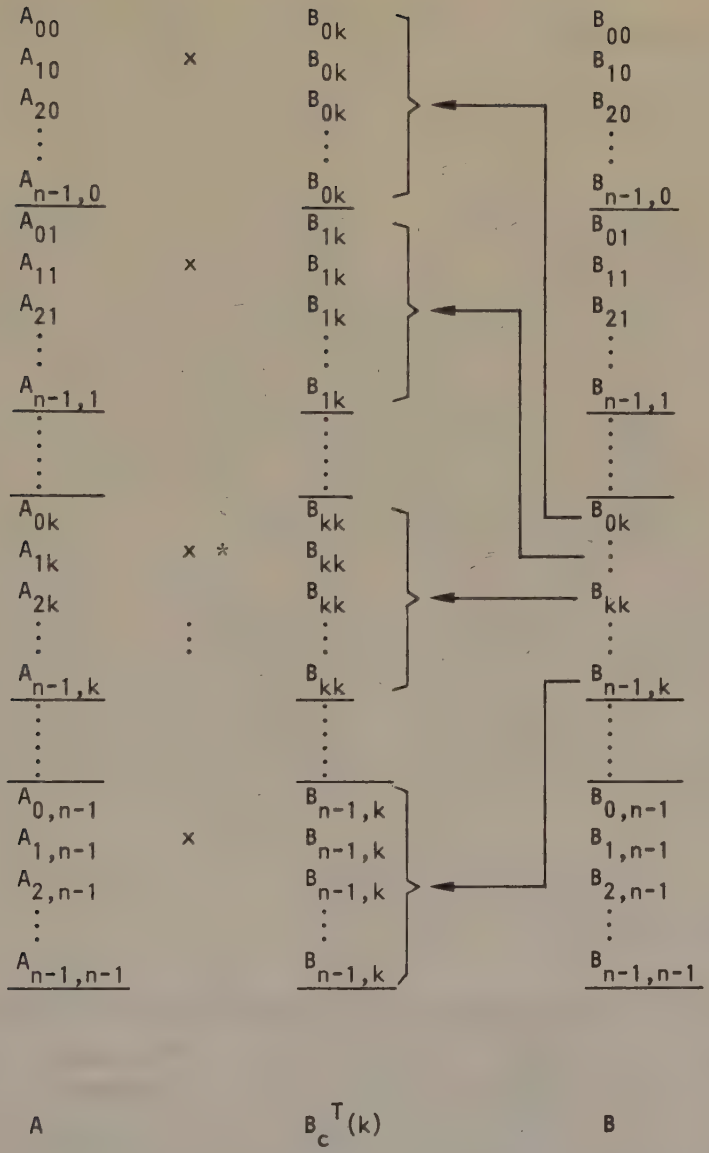


Figure 4.2.9 Alignment required for computation of column k according to the middle-product method. "x" mark rows where products contributing to element $R_{1,k}$ of the result matrix are computed. "*" marks the row where the element $R_{1,k}$ is to be stored.

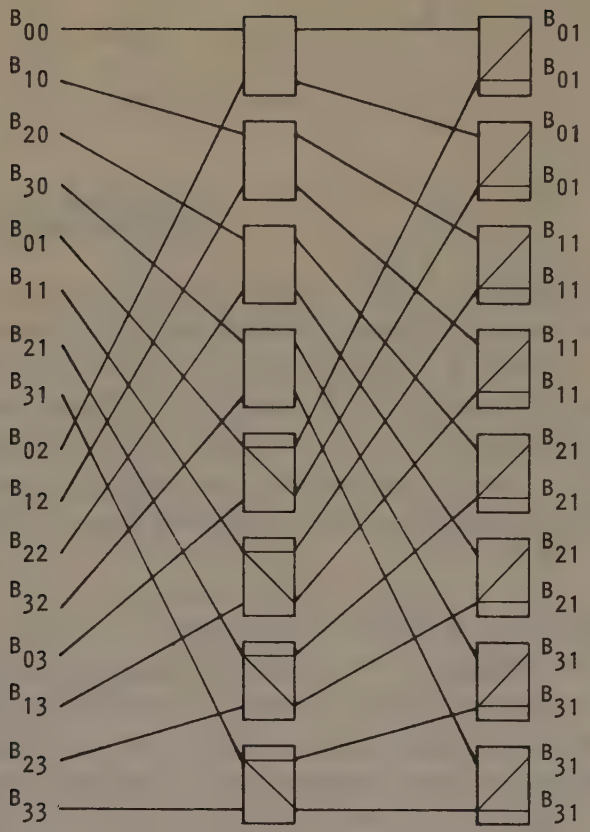


Figure 4.2.10 Formation of $B_c^T(1)$ from B.

The n products contributing to element no. i of this column are situated n words apart, starting at word i. The summation process needed to form the element can be done in parallel with those summations that form the other elements in the column. It is done using the perfect shuffle/exchange network in $\log_2 n$ steps, as depicted in Figure 4.2.13 a. Putting the result at the right destination requires $\log_2 n$ additional shuffles, which is shown in part b of the same figure. As can be seen however, these can be done in parallel for all columns.

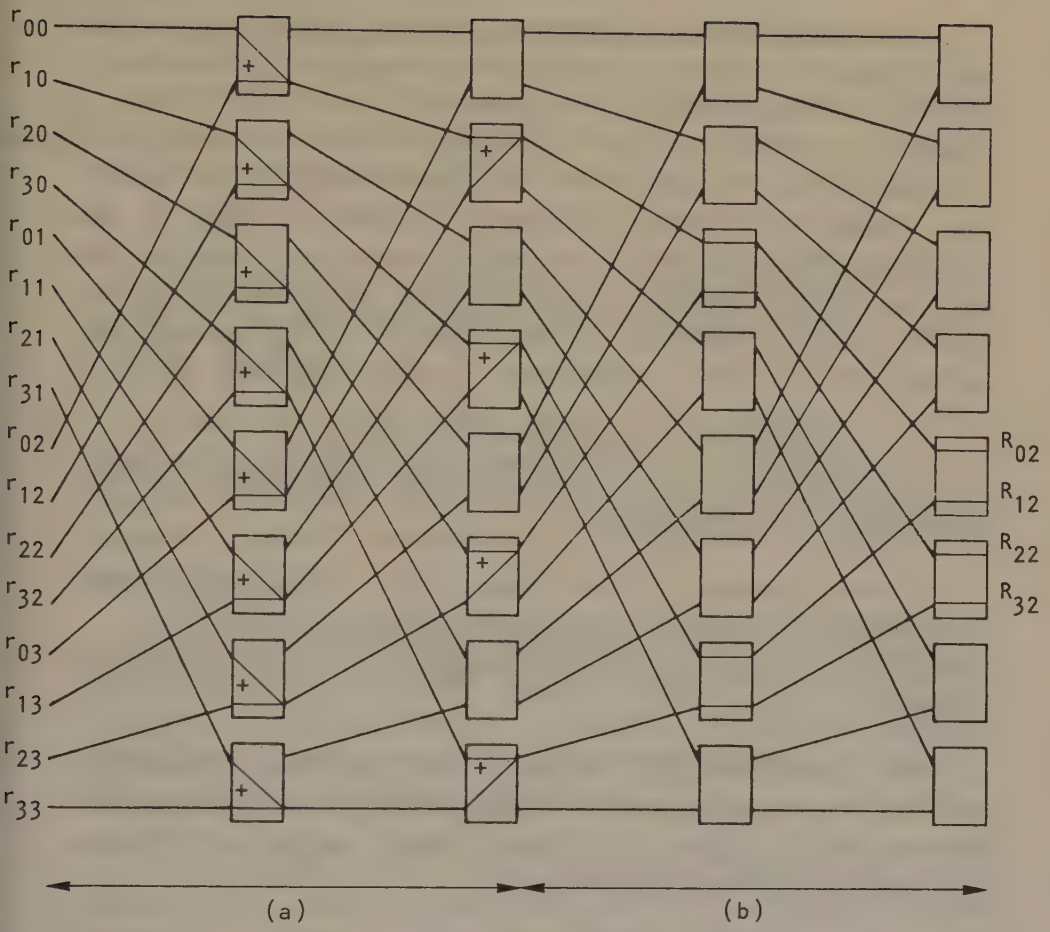


Figure 4.2.11 a) Computation of $R_0=r_{00}+r_{01}+r_{02}+r_{03}$
$R_1=r_{10}+r_{11}+r_{12}+r_{13}$
$R_2=r_{20}+r_{21}+r_{22}+r_{23}$
$R_3=r_{30}+r_{31}+r_{32}+r_{33}$

b) Putting the result at the right destination (column 2 assumed)

To sum up the amount of shuffling needed in this method, we note that neither the alignment procedure (the formation of B_c^T) and the addition procedure utilize the full parallelism if only one column is treated at a time. As was the case with the formation of A_c in previous method, all columns can be treated partly simultaneously. This reduces the number of stages to 2(n-1) for each of the procedures. Both broadcasting and addition require two reads and one write per pass. Therefore, the total number of passes through the shuffle/exchange network of LUCAS in order to align arguments for multiplication, add the contributions and rearrange the result is

$$P_2 = 2*3(n-1) + \log_2 n = 6(n-1) + \log_2 n,$$

where the $\log_2 n$ term comes from the final rearrangement of the result matrix (a transposition).

For n=8, P_2 amounts to 45 passes. The outer-product method described earlier, which uses computation in place, yields P_1 = 72 passes. Thus, it turns out to be advantageous not to use computation in place.

It is interesting to put the time for shuffling data in relation to the time for the multiplications. Approximating the time to pass the network with 3b cycles (b is the data length) and the multiplication time with $3b^2$ cycles yields, for n=8:

Shuffle time $P_2 = 45 * 3b = 135b$ cycles

Multiplication time $M = 8 * 3b^2 = 24\ b^2$ cycles.

b=8 gives P_2=1080 and M=1536 cycles. Thus, about 40% of the total time is spent for shuffling. With b=16, P_2=2160 and M=6144 cycles. In this case about 25% of the time is used for shuffling.

4.2.3 n x n matrices, more than n but fewer that n^2 processors

LUCAS with its 128 PEs does not fit into the scheme of Section 4.2.2, since 128 is not an even square. In this section we study the usefulness of the interconnection network when the number of PEs is e.g. $n^2/2$, or more general n^2/m, where m is a power of 2.

In such a case, the n^2 elements of a matrix are distributed over m fields in the associative memory. Figure 4.2.12 illustrates the case when n=4 and m=2.

We adopt the middle-product method. To align elements for the computation of column k, the matrix $B_c^T(k)$ is formed through broadcasts of column k. The total time to form all B_c^T matrices amounts to m*2(n-1) broadcasts if the procedure is parallelized maximally.

In each PE, m multiplications are made and the products added. The sums produced are added over the interconnection network in $\log_2(n/m)$ steps. This procedure of shuffling and adding can again be parallelized, to yield a total of 2n(m-1) addition steps. Finally, $\log_2 n$ shuffles of each of the m fields are made.

We see that the middle-product algorithm of Section 4.2.2 is well adopted to the case of fewer PEs. The full parallelism is used throughout the entire algorithm. This gives a processing time m times longer than if n^2 processors were available.

Matrix A		Part of matrix B	After broadcasting column 1	
A_{00}	A_{02}	B_{00}	B_{01}	B_{21}
A_{10}	A_{12}	B_{10}	B_{01}	B_{21}
A_{20}	A_{22}	B_{20}	B_{01}	B_{21}
A_{30}	A_{32}	B_{30}	B_{01}	B_{21}
A_{01}	A_{03}	B_{01}	B_{11}	B_{31}
A_{11}	A_{13}	B_{11}	B_{11}	B_{31}
A_{21}	A_{23}	B_{21}	B_{11}	B_{31}
A_{31}	A_{33}	B_{31}	B_{11}	B_{31}

Computations made in PEs	After addition over interconnection network	Results from column 0 and 1 merged	After 2 shuffles
$A_{00}B_{01}+A_{02}B_{21}$		C_{00}	C_{00}
$A_{10}B_{01}+A_{12}B_{21}$	C_{01}	C_{01}	C_{10}
$A_{20}B_{01}+A_{22}B_{21}$		C_{10}	C_{20}
$A_{30}B_{01}+A_{32}B_{21}$	C_{11}	C_{11}	C_{30}
$A_{01}B_{11}+A_{03}B_{31}$		C_{20}	C_{01}
$A_{11}B_{11}+A_{13}B_{31}$	C_{21}	C_{21}	C_{11}
$A_{21}B_{11}+A_{23}B_{31}$		C_{30}	C_{21}
$A_{31}B_{11}+A_{33}B_{31}$	C_{31}	C_{31}	C_{31}

Figure 4.2.12 Illustration of part of the computations when two 4x4 matrices are multiplied on an 8 PE array

4.2.4 n x n matrices, more than n^2 processors

We also briefly consider the case when there are more PEs available than there are elements of a matrix. For example, this is the case when 128 PEs are used for multiplication of 8x8 matrices.

Let there be $m*n^2$ PEs available, where m is a power of 2. The A- and B-matrices now each fill the upper n^2 words of a field. If full $m*n^2$ parallelism is to be utilized the matrix elements must be broadcast in a way that aligns them properly for multiplication.

If we spread column k of B as in Figure 4.2.10, we will automatically, with $m*n^2$ PEs, also spread columns k+1, k+2,...,k+m-1 to the rest of the field, provided that k is a multiple of m or k=0. After spreading A by means of $\log_2 n$ broadcasts and a sequence of shuffles, elements are aligned so that all multiplications needed for columns k, k+1,...,k+m-1 of the result matrix can be done simultaneously. After addition over the interconnection network, rearrangement of the result is needed in order to have the result matrix stored in the same order as the input matrices. This process is more complicated than in the earlier described cases. However, since it is done only once for the whole result matrix in parallel, the extra time caused by this is negligible.

A more detailed description of the $m*n^2$ case, as well as a more formal treatment of the n^2 PEs case, is given in (Ohlsson and Svensson, 1982).

4.3 FAST FOURIER TRANSFORM

The fast Fourier transform (FFT) is a method for efficiently computing the discrete Fourier transform (DFT) of a time series (discrete data samples). The DFT has properties that are analogous to those of the Fourier integral transform, which can be used to determine the frequency spectrum of a continuous, time varying signal. The publication of the FFT method (Cooley and Tukey, 1965) meant a revolution in signal processing, since the time needed to compute the DFT on a digital computer is reduced by orders of magnitude. A straightforward calculation of the DFT (according to the definition) on a sequential computer takes $O(N^2)$ time, where N is the number of samples, whereas only $O(N\log_2 N)$ time is needed when the FFT method is used. The algorithm is well suited for parallel processing. Using N processing elements, the processing time will be $O(\log_2 N)$.

First, we will give a short description of the DFT and the FFT algorithm. It is based on the description given in (IEEE G-AE, 1967), which can be consulted for further details. Then we will show how the FFT is implemented on LUCAS. The interconnection structure plays an important role in the computation.

4.3.1 The discrete Fourier transform

If a digital computer is to be used for analyzing a continuous waveform then it is necessary that the data be sampled. The minimal sampling rate needed in order to obtain a true representation of the waveform is twice the highest frequency present in the waveform.

Assume the time series obtained has length N. Denote by X_k the kth sample of the time series. The DFT of the time series consists of N complex coefficients, A_r, r=0,1,...,N-1. Each A_r is obtained by the formula

$$A_r = \sum_{k=0}^{N-1} X_k e^{-2\pi jrk/N} \qquad (4.3.1)$$

Using the shorthand notation $W = e^{2\pi j/N}$ the expression for A_r becomes

$$A_r = \sum_{k=0}^{N-1} X_k W^{rk} \qquad r=0,1,\ldots,N-1 \qquad (4.3.2)$$

The inverse of (4.3.2) is

$$X_k = (1/N) \sum_{r=0}^{N-1} A_r W^{-rk} \qquad k=0,1,\ldots,N-1 \qquad (4.3.3)$$

This relationship is called the inverse discrete Fourier transform (IDFT).

The DFT and the IDFT are of similar form, implying that a parallel machine suitable for computing one can be used for computing the other by simply exchanging the roles of X_k and A_r, and making appropriate scale-factor and sign changes. In fact, $IDFT(A_r) = (DFT(A_r)^*)^*$, where * denotes the complex conjugate.

4.3.2 The fast Fourier transform

The FFT is a clever computational technique to compute the DFT coefficients. The DFT of a time series is here obtained as a weighted combination of the DFTs of two shorter time series. These, in turn, are computed in the same way, until the DFT of a single point is needed. This is the point value itself, according to expression (4.3.2).

Suppose that the time series X_k, k=0,1,...,N-1, is divided into two functions, Y_k and Z_k, each of which has only half as many points. The function Y_k is composed of the even-numbered points $(X_0,X_2,X_4,...)$ and Z_k is composed of the odd-numbered points $(X_1,X_3,X_5,...)$, see Figure 4.3.1.

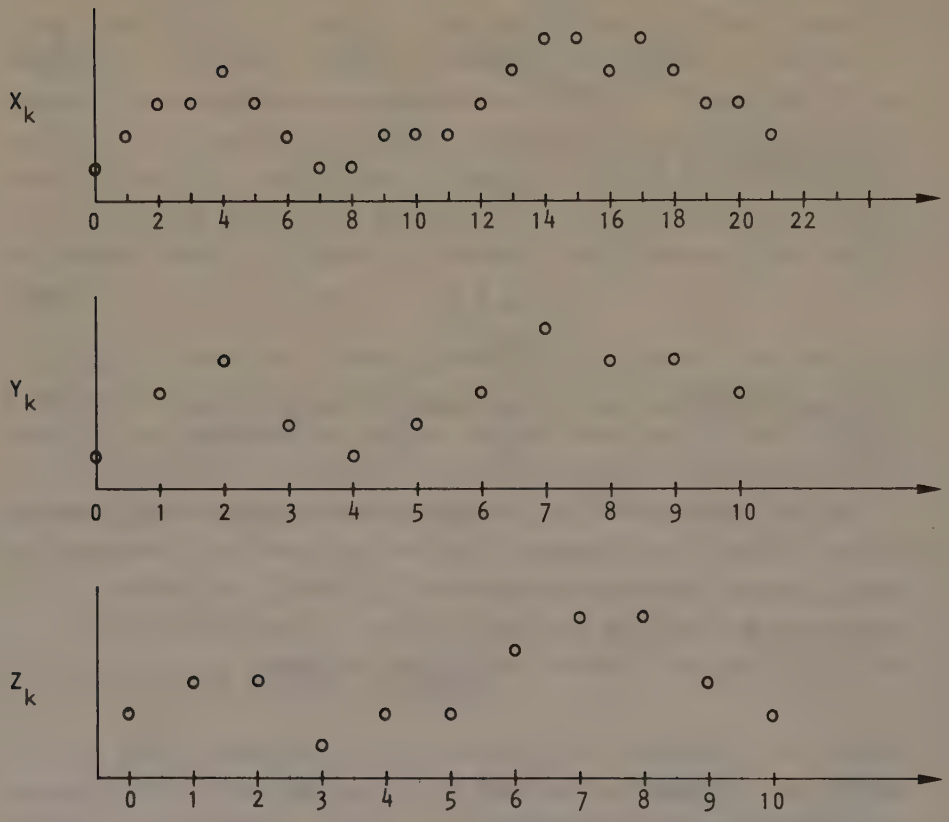


Figure 4.3.1 Decomposition of the time series X_k into two half as long series Y_k and Z_k

Now, if B_r and C_r denote the discrete Fourier transforms of Y_k and Z_k respectively, it is easily shown that the discrete Fourier

transform, A_r, of X_k can be written

$$A_r = B_r + W^r C_r \qquad 0 \leq r < N/2 \qquad (4.3.4)$$

$$A_{r+N/2} = B_r - W^r C_r \qquad 0 \leq r < N/2 \qquad (4.3.5)$$

From (4.3.4) and (4.3.5) the first N/2 and last N/2 points of the discrete Fourier transform of X_k (a sequence having N samples) can be easily obtained from the DFT of Y_k and Z_k, both sequences of N/2 samples. Figure 4.3.2 illustrates this for the case N=8.

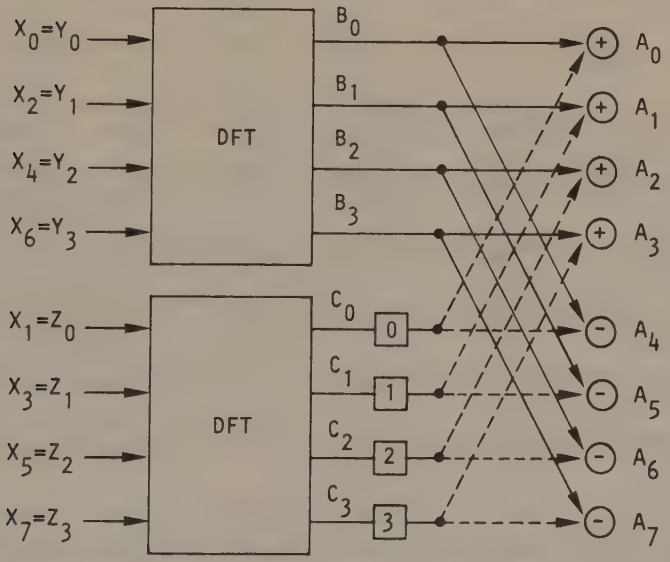


Figure 4.3.2 Signal flow graph illustrating how calculation of an 8-point DFT can be reduced to the calculation of two 4-point DFTs. A number within a square represents multiplication by W raised to the number. In the lower half, the value arriving by the dotted line is subtracted from the value arriving by the solid line. In the upper half the two values are added.

We can use this technique repeatedly, i.e. we can in turn divide X_k and Y_k into half as long sequences. Accordingly, the computation of B_k (or C_k) can be reduced to the computation of sequences of N/4 samples. These reductions can be carried out as long as each function has a number of samples that is divisible by 2. Normally, N is chosen to be a power of two. We will limit the discussion to that case.

The computation is illustrated by the signal flow graph in Figure 4.3.3.

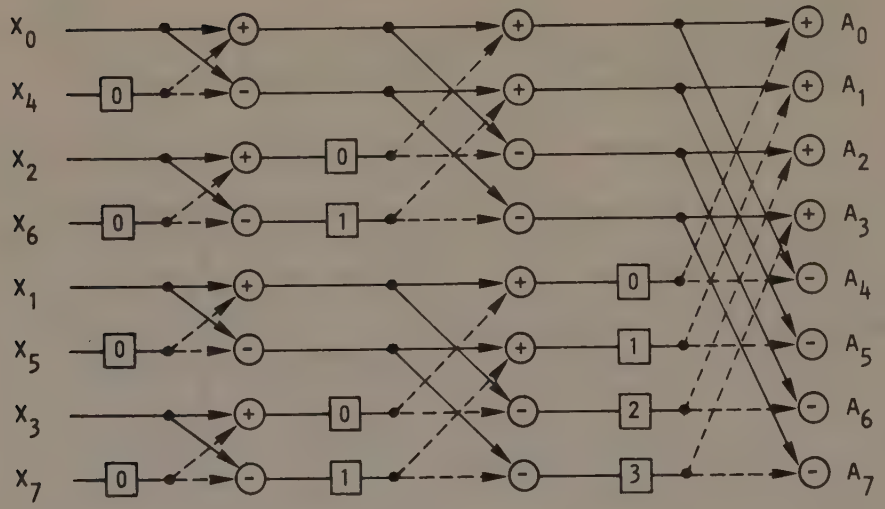


Figure 4.3.3 Calculation of an 8-point DFT using the FFT algorithm

The graph shows that the computation consists of $\log_2 N$ iterations. In each iteration, N/2 multiplications, N/2 additions and N/2 subtractions are made. Thus, a total of $(3/2)N\log_2 N$ operations are needed, one third of which are multiplications.

4.3.3 Implementation on LUCAS

In Figure 4.3.3, the coefficient values A_k are in conventional order, with increasing indices. The X_k samples, however, are in what is called "bit-reversed" order, meaning that the indices are in natural order if their binary representations are read backwards. If all the nodes on the same horizontal level as A_1 are interchanged with all the nodes on the same horizontal level as A_4, and all the nodes on the same level as A_3 are interchanged with all the nodes on the same level as A_6, with the arrows carried along with the nodes, the flow graph depicted in Figure 4.3.4 is obtained. In this graph, X_k is in the original order, whereas A_k is bit-reversed.

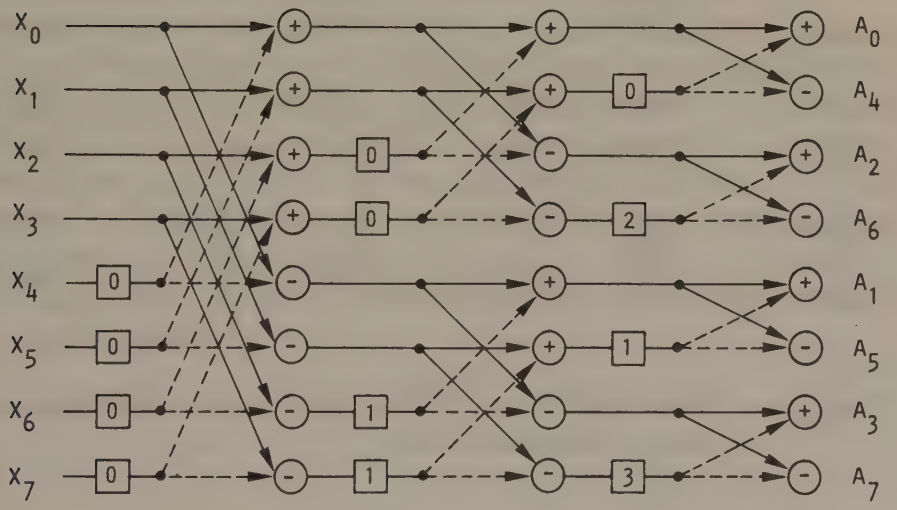


Figure 4.3.4 Rearrangement of the signal flow graph of Figure 4.3.3. X_k in natural order, A_k in bit-reversed order.

The interesting feature of this graph is that it has the same structure as the flip network of STARAN, run backwards, which we have shown to be equivalent to a shuffle/exchange network (see Section 1.3.3). Thus, the flow graph can again be re-drawn, using a shuffle+exchange structure in each iteration (Figure 4.3.5). This shows that a 128 point FFT calculation can be efficiently mapped onto LUCAS if the input data is placed one sample in each memory word.

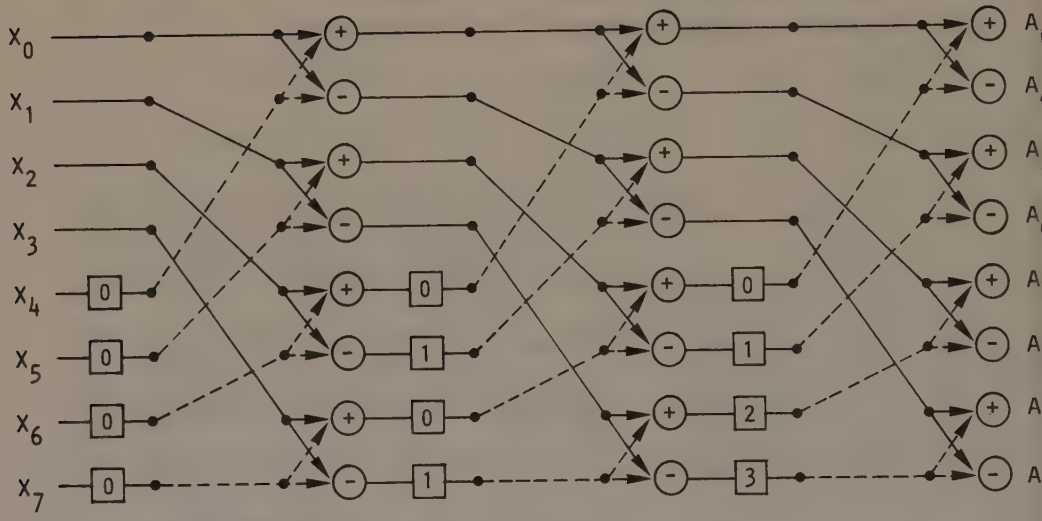
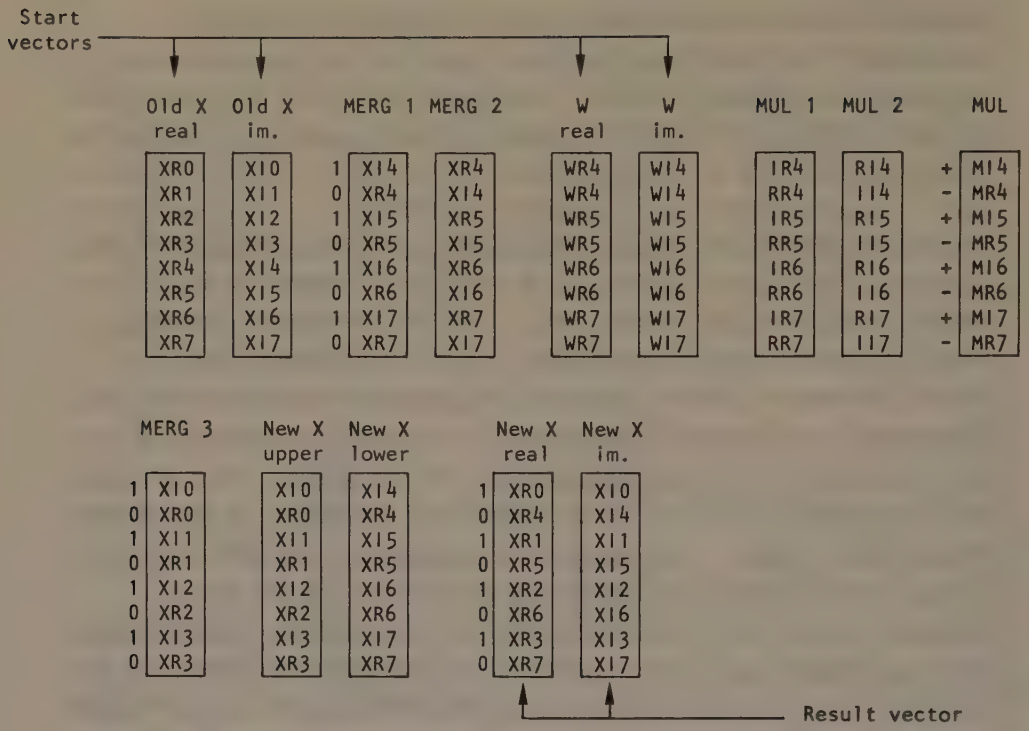


Figure 4.3.5 Adaptation of the FFT algorithm to the shuffle/exchange interconnection structure

The multiplications by the W-constants are the most time consuming calculations in the computations. It is unsatisfactory that only half of the PEs (the lower half) participate in these calculations, according to Figure 4.3.5. However, the quantities to be multiplied are complex numbers. Therefore a multiplication actually

consists of four multiplications, plus one addition and one subtraction. The real and imaginary parts of the quantities to be multiplied can be merged before multiplication in such a way that all the PEs participate in the multiplications. This reduces the number of multiplications from four to two.

Figure 4.3.6 helps to illustrate the detailed execution of one iteration. The starting point is a vector of complex X-values, in Figure 4.3.6 denoted "OldX", and a vector containing the complex W-values, each W-value appearing twice. The desired result is a vector of "NewX" values, appearing in perfectly shuffled order. The first step is to spread the lower half of "OldX" to all PEs in a way that is suitable for multiplication by the W-values. This is done through MERGE operations (see Chapter 2) resulting in the fields MERG1 and MERG2. The complex multiplication is done as two multiply-fields followed by addition in some words (giving the imaginary-valued part of the product) and subtraction in some (giving the real-valued part). This completes the multiplication phase of the iteration. The products thus obtained (the MUL field) are now to be added to and subtracted from the upper half of "OldX" in order to produce the new X-values. To this end, the upper half of "OldX" is spread out to match the MUL field. Addition and subtraction are performed and a final MERGE is needed to separate the real and imaginary parts of "NewX". "NewX" is obtained in perfectly shuffled order, as desired.



MERG 1 = MERGENS (Old X im, Old X real)

MERG 2 = MERGENS (Old X real, Old X im)

MUL 1 = MERG 1 · Wreal

MUL 2 = MERG 2 · Wim

MUL = MUL 1 $\pm$ MUL 2

MERG 3 = MERGENS (Old X im, Old X real)

New X upper = MERG 3 + MUL

New X lower = MERG 3 - MUL

New X real = MERGEBD (New X upper, New X lower)

New X im = MERGEDA (New X upper, New X lower)

Figure 4.3.6 Illustration of the execution on LUCAS of one iteration of the FFT

The entire algorithm has been written as a microprogram. The execution time for a 128 samples FFT when all data are 8-bit is 0.2 ms per iteration, making a total of 1.4 ms. The multiplications take 70% of the total execution time. Since the multiplication executes in a time proportional to the square of the data length, the ratio grows with increased data length. Noting that the real and imaginary parts of the coefficients in the first two iterations have the values zero and plus and minus one only, the multiplications in these iterations can be omitted. This reduces the execution time from 1.4 to 1.1 ms.

LUCAS can be used for computation of the FFT with full parallelity also when the number of samples does not match the size of the array. When the number is larger, e.g. 1024, samples that are 128 units apart are put in the same memory word. This makes it possible to compute the first iterations of the algorithm entirely within the PEs. Assuming 2^n*128 sample points, LUCAS will need $2^n(\log_2 128 + n)$ iterations of the kind described above to compute the FFT. The following table shows how this number and the execution time grows with the number of samples. (Reduction for the two initial iterations is made).

n	no. of samples	no. of iterations	time(ms)
0	128	7	1.1
1	256	16	2.6
2	512	36	6.0
3	1024	80	13.6
4	2048	176	30.4
5	4096	384	67.2

When the number of samples is smaller than the number of PEs, more than one FFT calculation can be performed at a time. For example, when the number of samples is 32, one sequence of 32 samples is put in memory words 0,4,8,..., another sequence in words

1,5,9,..., still another in 2,6,10,...,etc. The FFT of all four sequences can be calculated simultaneously.

As noted above, the result data from the FFT algorithm appears in bit-reversed order. To get the data out to the host computer in natural order in a simple manner, we have equipped LUCAS with an "address bit reversal" facility. The host can choose any of two buffers to pass the address to the I/O data registers of the processor array. One of the buffers transfers the address without any changes, the other buffer reverses the bits of the address. Thus data can be brought in or out in bit-reversed order by ordinary block moves.

LUCAS has been used for spectral analysis of speech in real time (Ohlsson, 1982). The sampling frequency needed in order to cover the significant frequencies of speech is 10 kHz. Real time analysis based on 128 samples requires that the computation be performed in 12.8 ms, which is the time to gather 128 samples. The 1.1 ms needed by LUCAS is well within this limit.

4.4 THREE GRAPH-THEORETIC PROBLEMS

Problems that can be identified as graph-theoretic show up in diverse areas, e.g. traffic planning and network analysis. A common task is to find the shortest path between two vertices of a graph. The connection between two vertices may be uni-directional or bi-directional. In the first case the graph is called a directed graph. Also, a cost (or path length) may be associated with each path. Such graphs are called weighted.

Solutions of problems of this kind often take the form of searching large trees or updating matrices. Opportunities to exploit the kind of parallelism offered by LUCAS are rich. As examples we will consider algorithms for the solution of two different shortest path problems on LUCAS. In the first problem, paths between vertices are all bi-directional and all have the length 1 (if they exist). The task is to determine the length of the shortest path between two specified vertices. In the second problem the paths are uni-directional and an individual length is associated with each path. The task is to produce a distance matrix showing the lengths of the shortest path between all pairs of nodes. We will also consider an algorithm for finding the minimal spanning tree of a graph, i.e. that subset of edges of the graph that connects all vertices with minimal total edge weight.

4.4.1 Shortest path between two given vertices. Unit path length

Figure 4.4.1 shows a graph that we will use as an example to illustrate the proposed algorithm. From each vertex, lines are drawn to vertices that can be reached directly, i.e. with path length 1. A compact way of representing the graph on LUCAS is by means of an 'adjacency matrix' shown in Figure 4.4.2. A '1' in the matrix indicates that there is a direct connection between the vertices in the row and column. Since all paths are bi-directional, the matrix is symmetrical around the main diagonal. In LUCAS the matrix is stored one row per memory word, one column per bit-slice.

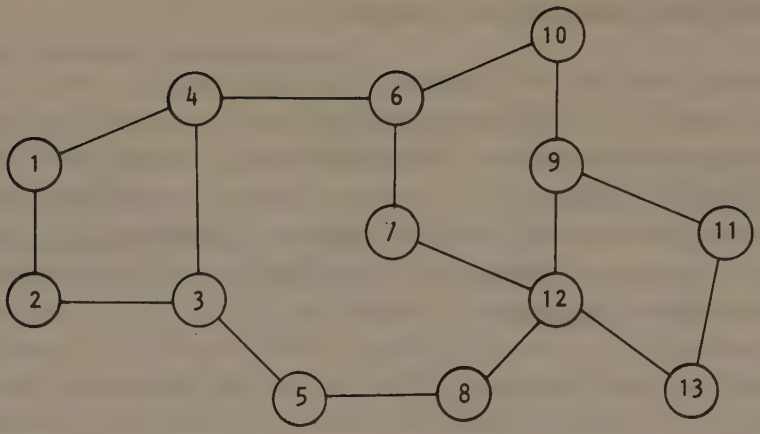


Figure 4.4.1 A bi-directional graph

As an example, we want to find the length of the shortest path between vertex no. 2 and vertex no. 11. We do this by successively building the tree of vertices reachable from 2 in one, two, three,... steps. To start with, the vertices that can be reached in one step are marked in the Tags. OR-ing the contents of those rows that now are tagmarked, gives a row, a "mark word", indicating vertices reachable in exactly two steps. In the next step the logical OR of all bit-slices marked in the mark word is formed and the result is stored in the Tags. The tags now indicate which vertices can be reached in three steps. This procedure is continued, alternating between horizontal and vertical OR-ing of marked bit-slices and rows, respectively, until finally, the destination vertex is reached. In this case we arrive at the destination vertex after six steps.

In each iteration the entire matrix has to be traversed bit-slice by bit-slice. Thus, the time to perform one iteration is proportional to the number of vertices, n. The number of iterations is the same as the length of the shortest path, l.

Execution time = constant * l * n

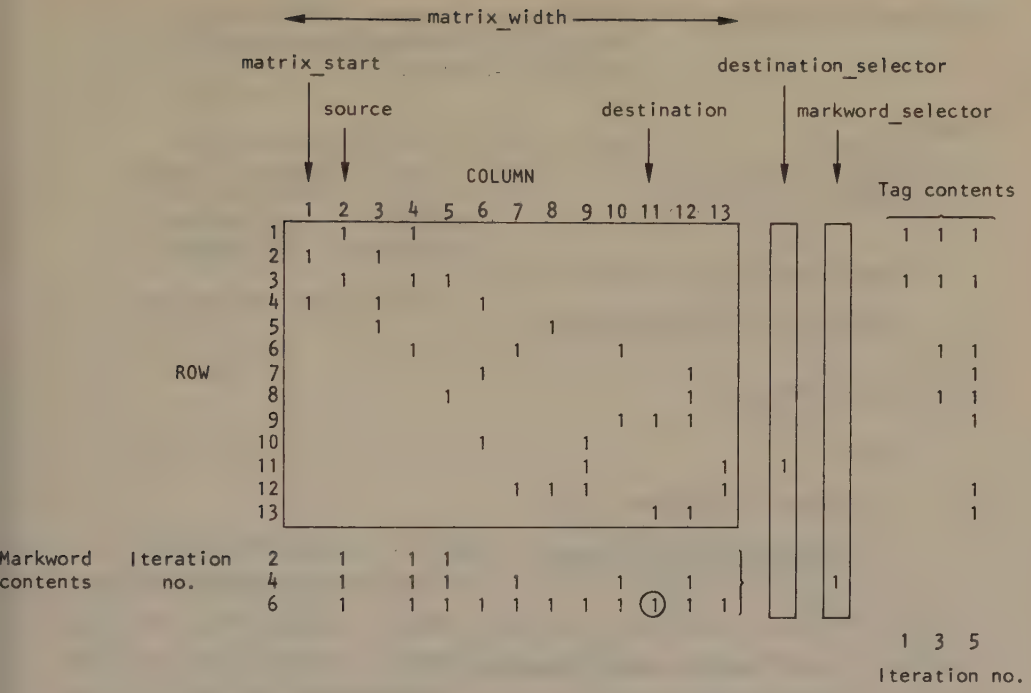


Figure 4.4.2 Adjacency matrix of the graph in Figure 4.4.1. Pointers administrated by the Address Processor are shown, and also the changing contents of the Tags and Mark word as the algorithm proceeds

We will now make the description more precise by giving it in the high level microprogramming language. The number of parameters needed in the instruction is 6. Thus, the parameters must be loaded to the Control Unit in two passes. One extra register of the Address Processor is needed to count the number of iterations.

```
Microprogram SHORTPATH  {matrix_start, matrix_width, source,
                         destination, destination_selector,
                         markword_selector}

Begin
     counter:=1;
                                    (* Clear iteration counter *)
     LTMA(source,direct);
                                    (* Mark in Tags vertices
                                    reachable in one step *)
     ANDTMA(destination_selector,direct);
                                    (* See if dest reached *)
     If SOME then exit(SHORTPATH);

LOOP: While TRUE do

      Begin

        counter:= counter+1; x:=0;

        Iterate matrix_width times
                                    (* For each bit-slice, *)
        Begin

          LTMT(matrix_start+x,direct); CRA;
                                    (* form logical OR *)
          if SOME then CORA;
                                    (* of marked words *)
          LTMA(markword_selector,direct);

          WRRT(matrix_start+x);
                                    (* Write result in markword *)
          x:= x+1;

        End;

        LTMT(destination,direct);
                                    (* Check if dest reached *)
        if SOME then exit(LOOP);

        counter:= counter+1; x:= 0;

        CRA;

        Iterate matrix_width times
                                    (* Each bit-slice *)
        Begin

          LTMA(markword_tag,direct);
                                    (* marked in mark word *)
          LTMT(matrix_start+x,direct);
                                    (* contributes to *)
          if SOME then ORRMA(matrix_start+x,direct);
```

```
                                        (* horizontal OR *)
        End;

        LTRA;

        ANDTMA(destination_selector,direct);
                                        (* Check if dest reached *)
        if SOME then exit(LOOP);

        LTRA;

      End;

End;
```

From the information in Figure 4.4.2, gathered during the computation, it is possible to trace back which route or routes that give the shortest path length. Logical AND between bit-slice no. 11 and the Tag contents from iteration no.5 gives '1's at rows 9 and 13. Thus, there are two paths to 11, one via 9 and the other via 13. Now, ANDing row 9 and the mark word from iteration no. 4 gives that the path that passed 9 goes via 10 or 12, etc. To be able to perform this back-tracking we see that successive mark word and Tag contents must be saved. This is easily done and adds very little to the total execution time.

4.4.2 Shortest path between all pairs of vertices in a weighted, directed graph

In a weighted, directed graph the paths between vertices are uni-directional and there is a length associated with each path. Figure 4.4.3 shows an example of such a graph. We will consider the problem of finding the shortest path between all pairs of vertices. The graph is given in the form of a matrix. The matrix corresponding to the graph in Figure 4.4.3 is given in Figure 4.4.4. Note that the absence of a direct path between a pair of vertices is marked

"infinite" (if) in the matrix.

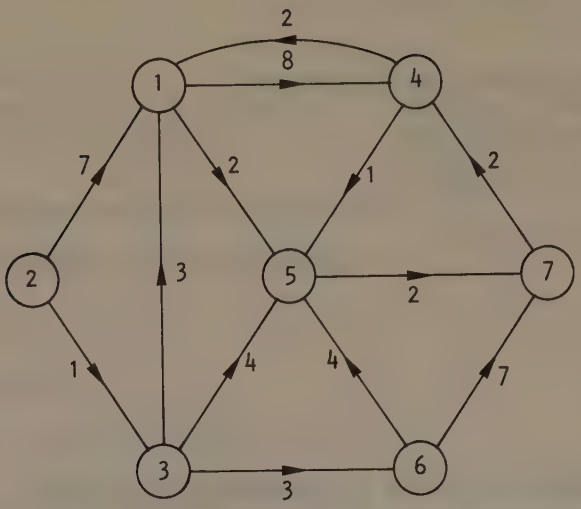


Figure 4.4.3 A weighted, directed graph

	1	2	3	4	5	6	7
1	0	if	if	8	2	if	if
2	7	0	1	if	if	if	if
3	3	if	0	if	4	3	if
4	2	if	if	0	1	if	if
5	if	if	if	if	0	if	2
6	if	if	if	if	4	0	7
7	if	if	if	2	if	if	0

Figure 4.4.4 The distance matrix of the graph in Figure 4.4.3

To solve the problem on LUCAS we will follow an algorithm due to Floyd (1962), which is considered as one of the two most efficient algorithms for sequential computers. It is well suited for parallel implementation. On sequential computers a computation time

proportional to n^3 is required, where n is the number of vertices. On a parallel computer with n PEs it should be possible to perform the algorithm in a time proportional to n^2.

The algorithm works as follows. Starting with the original n by n matrix D of direct distances, n different matrices D_1, D_2,..., D_n are constructed sequentially. Matrix D_k is obtained from matrix D_{k-1} by inserting vertex k in a path wherever this results in a shorter path.

On a parallel computer with n PEs an entire column of the matrix can be updated simultaneously. In the k:th iteration, column p of D_k is obtained in the following way (using Pascal/L notation for matrix elements):

$$D_k(,p) := \min [\, D_{k-1}(,p)\ ,\ D_{k-1}(,k) + D_{k-1}(k,p)]$$

A Pascal/1 program for the entire algorithm reads as follows:

```
Program FLOYD;

const noofvertices = 128;
var Dmatrix: parallel array [1..noofvertices,1..noofvertices] of
integer(8);
    k,p: integer;

begin
  for k:=1 to noofvertices do
  begin
    for p:=1 to noofvertices do
    begin
      where (Dmatrix[*,k]+Dmatrix[k,p]) < Dmatrix[*,p] do
      Dmatrix[*,p]:= Dmatrix[*,k]+Dmatrix[k,p];
    end;
  end;
end.
```

It is easily seen that the execution time of this program is proportional to n^2. The task that is performed n^2 times is an 'add fields' instruction followed by 'field larger than field' instruction and a tagmasked 'move field'. These are all proportional to the field length.

The algorithm requires a representation for an infinite value. We choose a number that is a little smaller than half the largest value that is possible to represent in the given field length. In the worst case, two such numbers are added. This will give no overflow.

4.4.3 Minimal spanning tree

The minimal spanning tree (MST) of a weighted, bi-directional graph is defined as that subset of the vertices of the graph that connects all vertices with minimal total edge weight. As an example of a context where the problem of finding the MST arises, consider a telecommunications system. The problem of connecting a set of cities to each other using minimal wire length is exactly the problem of finding the MST (provided that only wires from one city to another are allowed).

An efficient algorithm for finding the MST of a graph is due to Prim (1957). It was improved and implemented on computer by Dijkstra (1959) and is normally called the Prim-Dijkstra algorithm. On sequential computers it requires a processing time proportional to n^2 on an n-vertex graph.

The algorithm works by successively expanding a subtree (called a fragment) until, eventually, a spanning tree is obtained. The initial fragment consists of a single vertex, which may be chosen arbitrarily. The fragment is then expanded at each stage by adding to

it the nearest neighbour of the fragment, i.e. that vertex not in the fragment with minimal distance to the fragment. Ties are resolved arbitrarily. After n-1 stages the MST has been constructed.

As an example, consider the graph shown in Figure 4.4.5. Starting with vertex B in the fragment, edges are added to the subtree in the following order: B-D, D-A, B-C, C-E, E-F.

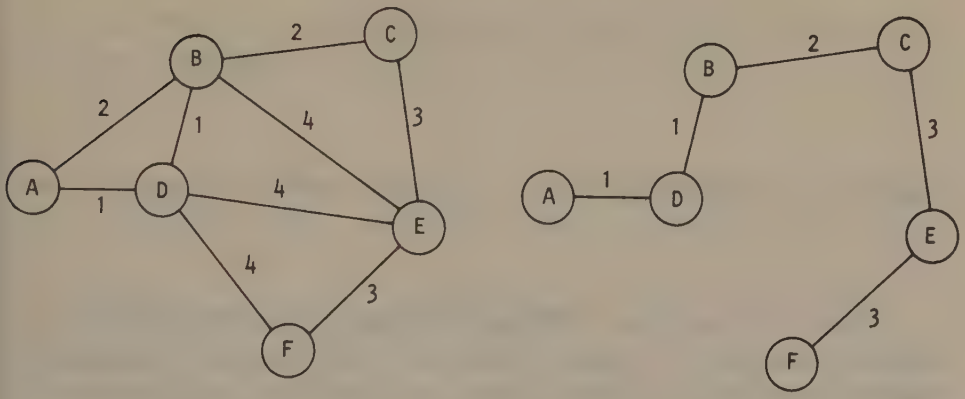


Figure 4.4.5 Weighted, bi-directional graph (left) and its minimal spanning tree (right)

To implement the algorithm on an n-processor array, we use the same representation of the graph in the associative memory as in the all-to-all shortest path problem above, i.e. a distance matrix. The distance matrix of our example graph is shown in Figure 4.4.6.

	A	B	C	D	E	F
A	0	2	if	1	if	if
B	2	0	2	1	4	if
C	if	2	0	if	3	if
D	1	1	if	0	4	4
E	if	4	3	4	0	3
F	if	if	if	4	3	0

Figure 4.4.6 Distance matrix of the graph shown in Figure 4.4.5

In order to determine which vertex to add to the fragment, Prim's original algorithm is keeping track of the "nearest nonfragment neighbour" of every fragment vertex. The algorithm then requires a running time proportional to n^3. Dijkstra's improvement resulted from using another strategy: keeping track of the "nearest fragment neighbour" of each nonfragment vertex. This gives $O(n^2)$ processing time.

Dijkstra's strategy turns out to be the most favourable also on a parallel processor like LUCAS. A distance table is needed in each stage. It contains, for each nonfragment vertex, the name of its nearest neighbour in the fragment and the distance to it. For example, after stage 2, when the fragment consists of vertices B, D and A, the distance table has the following contents:

nonfragment vertex	Nearest neighbour in fragment	Distance
C	B	2
E	B	4
F	D	4

C is chosen as the new fragment member and the table gets the following contents:

E	C	3
F	D	4

When implementing the algorithm on LUCAS we must make sure that all required information passing within the machine can be done. Figure 4.4.7 shows the contents of the distance table after each stage. After the search for minimum value of the distance column (D), and a SELECT FIRST to resolve ties, information about which vertex was chosen must be passed to the Address Processor. This information is used by the Address Processor to address that column of the distance matrix that should be merged into the D column of the distance table on the basis of "smallest value wins".

After interation no: 0

Vertex label	NN	D	T
A	B	2	1
B	B	0	0
C	B	2	1
D	B	(1)	1
E	B	4	1
F	B	if	1

After interation no: 1

Vertex label	NN	D	T
A	D	(1)	1
B	B	0	0
C	B	2	1
D	D	0	0
E	B	4	1
F	D	4	1

After interation no: 2

Vertex label	NN	D	T
A	A	0	0
B	B	0	0
C	B	(2)	1
D	D	0	0
E	B	4	1
F	D	4	1

After interation no: 3

Vertex label	NN	D	T
A	A	0	0
B	B	0	0
C	C	0	0
D	D	0	0
E	C	(3)	1
F	D	4	1

After interation no: 4

Vertex label	NN	D	T
A	A	0	0
B	B	0	0
C	C	0	0
D	D	0	0
E	E	0	0
F	E	3	1

To fragment:	B	D	A	C	E	F
Output:	B	B-D	D-A	B-C	C-E	E-F

Figure 4.4.7 The successive contents of distance table and tags

The 'Vertex label' field is necessary for this passing of addresses. It contains, for each vertex, the address of its column in the distance matrix. (Actually the field may contain only the binary representation of the word number. This number is then passed to the Address Processor where it is shifted left a few positions and added to an offset address to get the actual column address). The transfer of a data value from the associative memory to the Address Processor is made via the I/O register and the I/O Buffer Register.

At each stage of the algorithm the following is done:

* A search for the minimum value of the D-column in selected words.

* Output of the pair <NN,Vertex label> of the selected word.

* Transfer of the contents in the 'Vertex label' field of the selected word to the Address Processor, and spreading this label to a scratch pad field.

* Merging a new column from the distance matrix into the D-column on the basis of "smallest value wins".

* Merging the scratch pad field into the NN-field using the merging mask determined above.

All these tasks take a time that is independent of the number of vertices, but proportional to the lengths of the fields they work on, in most cases the number of bits in the distance values.

The number of stages in the algorithm is n-1. Thus, we conclude that the task of finding the minimum spanning tree on LUCAS grows only linearly with the number of vertices, provided that this number is smaller than or equal to the number of processing elements. In fact, linear time is optional if the list of edges is to be output in series. This is because each new edge means adding a new vertex to the subtree, and the number of vertices to be added is n-1. We have shown that the decision which new vertex to choose can be taken in constant time.

4.4.4 Discussion

We have demonstrated how three frequently encountered graph theoretic problems can be solved efficiently on LUCAS. The latter two show an improvement with a factor n compared to sequential execution. As for the first one, an algorithm due to Dijkstra (1959) solves the problem in a time proportional to n^2 on a sequential computer. The time on LUCAS - using an entirely different algorithm - is proportional to n*l, where l is the length of the shortest path. Preliminary studies give at hand that Dijkstra's algorithm can be

followed also in implementation on LUCAS. It would have some characteristics in common with the MST algorithm. The algorithm presented here in part 4.4.1 is simpler to follow and program but probably less efficient if the number of vertices is large and the shortest path between the specified vertices is long.

The parallel implementation of Floyd's shortest path algorithm is very straightforward. The MST algorithm, however, is more tricky. We have not seen any reports on the solution of the MST problem on an n-processor computer. Bentley (1979) describes an implementation of the Prim-Dijkstra algorithm on an $n/\log_2 n$-processor tree-structured system. The execution time is $n\log_2 n$, which means that he is also able to use the full parallelism.

Graph theory is an area with many problems open for parallel solution. With the examples given we have indicated that parallel processors with architectures similar to LUCAS bear good promise to be useful for these purposes.

CHAPTER 5
IMAGE PROCESSING

Our concern in this chapter is to investigate the usefulness of LUCAS - and LUCAS-type of processors - in image processing. We first state some demands that are put on a computer for image processing. We then review earlier attempts to meet these demands through the use of unconventional computer architectures. We give arguments that the kind of parallelism offered by LUCAS is the one that is most useful for some very important image operations. Next we discuss different organizations of a processor array giving arguments for and against different structures. We also treat the question of how to best map an image onto a certain processor structure. Section 5.4, which constitutes the main part of the chapter, describes how image operations are performed on LUCAS organized as a linear array of processing elements. The main concern are operations that can be performed by local processing, but also measurements and global transforms are treated. The investigation is carried out in the form of examples. Timings are given and comparisons are made both with a conventional computer and with special purpose image processors.

5.1 COMPUTATIONAL DEMANDS IN IMAGE PROCESSING

5.1.1 Introduction

Image processing is characterized by large amounts of low precision data. A typical image size is 512 by 512 picture elements (pixels) of 8-bit data, i.e approximately 2 million bits. On the other hand, image processing offers possibilities to treat very many data

items in parallel.

The image processing area is usually divided into image analysis and enhancement on the one hand and image coding on the other. As for the first issue, a pattern recognition task calls for analysis of a picture leading to a description of it in terms of features. The computation of features normally involves a variety of picture to picture transformations. Many of these transformations are useful in their own right as image enhancement operations.

The purpose of image coding is to compress the information in an image as much as possible. A key notion is tranformation of pictures to a form in which they are represented by less correlated data. In this representation less significant data may be removed without too much distortion being introduced. The major tools in this area are reversible linear transforms. LUCAS and similar processors are useful also for this task. However, we have not analysed this, and we will not further consider the image coding area in this text.

Image enhancement and image analysis often takes place in cooperation between a human operator and a computer. This is frequent in e.g. medical applications. The operator takes the decisions which operations should be performed, the machine does the calculations. For example, the operator may call for an edge extraction operation to be performed, then point to an edge point in the image and ask the system to mark all edges connected to that point. Finally, the operator may ask for the areas of the different segments now present in the image.

This kind of interactive use calls for processing times not longer than, typically, one second in order not to be disturbing. For most operations these demands are not met by ordinary computers.

Also in completely automatic image analysis - without human interaction - strong demands are often put on the processing time. When analysing medical samples or samples of materials it is desirable to be able to analyse as many samples as possible in as short time as possible. Normally the desire is to increase the

throughput of data compared to human analysis.

Furthermore, there is a constantly increasing interest in using pictures as input information in automatic control and manufacturing. If we consider how handicapped a man would be in many such activities if he was not allowed to use his eyes, we understand that this is an attractive path of development. The dynamics in the controlled process puts certain constraints on the time available for the necessary analysis of the input picture. 100 ms for a relatively advanced analysing task may serve as a typical example.

5.1.2 General demands on a computer for image processing

Much image processing has been done on conventional computers. Furthermore, many special purpose designs have been made, each aimed at speeding up specific processing tasks. Thus, experience is quite large on the desirable characteristics of an image processor, and the effects of different approaches to performance enhancement. A list of desirable features of a computer designed for image processing may include the following points:

* It should not only provide a fixed set of operations but should be programmable at a level that allows new algorithms to be implemented and tested.

* It should be able to handle efficiently many different image transformation tasks, ranging from simple operations on binary pictures through gray scale modifications and thresholding to complex filters and global transforms.

* It should be able to handle efficiently different kinds of feature extraction tasks.

* It should provide very high efficiency in those tasks that are identified as the most frequent ones.

* It should be able to cope with widely varying image sizes and number of bits per pixel. Desirable is that the same program can be used for different values of these parameters.

* It should have an efficient input/output facility. The time for input/output should stand in reasonable proportion to the computation time.

These demands all have a fuzzy meaning, the last two probably being those that are more precise. However, they serve the purpose of being a kind of check list when different computer designs are considered as general purpose image processing machines.

5.2 DIFFERENT ATTEMPTS TO MEET THE DEMANDS

Due to the constantly increasing importance of the image processing field we see a growing number of special purpose processors built to meet the computational demands. The first proposal for a special purpose computer for image processing is due to Unger (1958). However, not until the late sixties and early seventies machines were actually implemented. We will divide some of the implemented machines into different categories, based on the principles of organization. A more elaborate overview can be found in (Danielsson and Levialdi, 1981). Many of the processors are described by their designers in (Duff and Levialdi, 1981).

5.2.1 Fast neighbourhood access

The importance of local operations in image processing was early understood, as was the discrepancy between the picture geometry and the linear memory space of a conventional computer. Computing the addresses of neighbouring pixels took too much time.

Picap I (Kruse, 1973), one of the first picture processors to be

built, uses two 61-stage shift registers in order to provide parallel access to the complete 3 x 3 neighbourhood of each pixel. The pixels are operated on sequentially and the picture size is fixed to 64 x 64 4-bit pixels. Picap I has two special purpose processors, one for logic operations and one for linear filters.

5.2.2 A small number of special purpose processors

Some recent designs use a small number of identical, carefully designed, special purpose processors working in parallel. An example of this is the filter processor FIP (Kruse et al.,1980) developed at Linkoping University, Sweden. The FIP processor is incorporated in a larger system, Picap II, containing many special purpose processors. FIP is used for the low level image to image transformations. Other processors serve e.g. input/output and image management functions.

Another system designed at Linkoping University incorporates another processor of this category, the GOP processor (Granlund, 1981). GOP and FIP each use four parallel subprocessors. The subprocessors, in turn, use pipelining. The organization of the GOP subprocessors is strongly adapted to the nature of a certain general operator type. An important feature of the FIP processor is the ability to reach the elements of an almost arbitrary sized neighbourhood very fast. This is accomplished through a quite large cache memory (32 kbyte) holding a portion of the picture.

In both these machines all subprocessors perform the same operations, thus operating in a SIMD manner.

Another processor of this type - also called FIP - is included in the FLIP system developed in Karlsruhe, West Germany (Gemmar et al, 1981). The 16 identical processors in the FLIP-FIP can work in either MIMD mode or SIMD mode. Each processor has its own program memory and instruction decoding circuitry. The processors may be arranged according to the topology of the processing task.

The Picap-FIP, GOP and FLIP-FIP implementations show that carefully designed special purpose processors can give considerable prestanda although very few processors work in parallel.

5.2.3 A large number of conventional microprocessors

During the last years designs using a large number of standard, conventional microprocessors have been proposed. The most well-known are the ZMOB (Rieger et al, 1980) and PASM (Siegel, 1981) systems. The number of processors used in these systems is in the order of the squareroot of the number of pixels in a picture.

With a large number of processors, the design of the interconnection structure for communication between them becomes a critical issue. ZMOB and PASM use radically different structures for interprocessor communication. ZMOB, with its 256 Z80 microprocessors, uses what is called a conveyor belt. This is a 257 stage, ring formed 8-bit wide shift register with one stage ('mailbox') for each processor and one for the host computer. The PASM system with 2^n processors (typically 1024) is planned to be equipped with an n-stage interconnection network. Considered in particular are "the generalized cube" and "the augmented data manipulator". Networks of this kind provide very rich possibilities for data exchange between processors (Siegel and Smith, 1978). For example, an arbitrary permutation can be performed in three passes through the network (Parker, 1980).

An important feature of the networks considered for PASM is their partitionability. This means that the system can be configured to work as several subsystems of parallel machines, each subsystem controlled by its own program. This is called multiple SIMD mode if the individual processors in each subsystem are controlled by the same program. If each processor runs its own program, which is also possible, the total system is said to work in multiple MIMD mode.

5.2.4 A very large array of simple processors

A major part of the computational burden in image processing is image to image transformations using operations of local nature. All new pixel values can be calculated independently of each other, using only the old pixel values in a small neighbourhood as arguments.

Considering this fact it becomes attractive to arrange a large number of processors in a two-dimensional structure. Two special purpose processors for image processing designed along these lines are CLIP4 designed at University College, London (Duff,1979) and MPP designed at Goodyear Aerospace, Ohio under contract from NASA and particularly intended to process satellite imagery at high speed (Batcher,1980). The same structure is also used in the general purpose processor array DAP, a commercial product from ICL, England (Flanders et al, 1977).

An overview of these designs was given in Chapter 1. All three machines use an array of bit-serial processors implemented in LSI. The size of the CLIP4 array is 96 x 96 processors. MPP has a 128 x 128 array while DAP has been implemented in both 32 x 32 and 64 x 64 array versions. The processors are controlled by a central control unit which provides identical control signals and memory addresses to all PE's. The control unit in turn gets its instructions from a host computer of conventional type.

Each processor in DAP and MPP is connected to its four nearest neighbours (north, south, east, and west). In CLIP4 also diagonal neighbours are connected resulting in eight directly connected processors.

The size of the data memory associated with each processor is for MPP typically 1 kbit and for DAP typically 4kbit. CLIP4 has a very small memory associated with each processor - only 32 bits - which is a severe limitation, especially in grey scale and colour image processing.

5.2.5 LUCAS compared to other machines

Concerning the similarity of LUCAS to the machines mentioned above, we note the following:

LUCAS is a SIMD computer composed of bit serial processing elements like MPP, CLIP4 and DAP. The number of processing elements is, however, about two orders of magnitude less than that of the above mentioned machines. This has consequences for the way pictures are best stored and manipulated. This, in turn, determines which interconnection structure between individual processors is the most suitable.

The number of processing elements is more like that of PASM and ZMOB but the processors are very different. The processors in LUCAS have no instruction decoding and sequencing circuitry. They can only work in SIMD mode. Using fully equipped microprocessors like in PASM and ZMOB is of course attractive considering the very low cost to which these can now be achieved. However, if they are primarily intended for SIMD use - which also PASM and ZMOB apparently are - too much redundant circuitry will be present in the system. Taking into consideration a future development with integration of several processors into a single chip, the use of processing elements without redundancy is a better way to follow. Availability of large scale integration technology is increasing tremendously, which makes this an important aspect.

5.2.6 The advantages of image parallelism

The different dimensions of parallelism that are open for utilization in image to image transformations of local nature (neighbourhood operations) are lucidly identified in (Danielsson and Levialdi, 1981). The four possibilities are:

-Operator parallelism: The sequence of operations to be performed on the image according to a chosen algorithm is performed in parallel

in a pipelined fashion.

-Image parallelism: Several pixels of the image are treated in parallel using multiple processing units.

-Neighbourhood parallelism: The processor has access to all neighbourhood pixel values simultaneously.

-Pixel bit parallelism: The bits in a pixel are treated in parallel. This is the only dimension of parallelism utilized in a conventional computer.

The range of parallelism in each of the four dimensions is between one and in the order of a hundred for all dimensions but the image coordinate dimension. Here the range is from a few thousand up to several million image points. Investments in image parallelism can almost always be utilized. This is not the case with the other types.

According to this, a processor of LUCAS type has potential to be efficient for local image operations. The main concern of this chapter is to investigate this by programming and timing several algorithms on LUCAS. To be really useful, a processor array of a certain size should be able to efficiently handle images of different sizes. This will be treated briefly. We will also consider some other types of operations, generally regarded as important, e.g. global transforms and image measurements. Input/output of pictures to/from a processor array is also an important issue.

After having decided to use image parallelism only, there are still degrees of freedom concerning the interconnection of the processing elements and the mapping of images to the processor structure. We will devote one section to a discussion of these topics, before presenting the investigations in Section 4.

5.3 ORGANIZATION OF PROCESSOR ARRAYS FOR IMAGE PROCESSING

5.3.1 Introduction

Processor arrays designed to utilize image parallelism can be configured in many different ways. Besides different arrangements of the interconnection between processing elements the mapping of images to this structure can be made in different ways. Given an interconnection structure, one mapping may be favourable for certain operations, another mapping for other operations. A question of great concern is also which mapping is the most favourable for input/output. Often, different mappings require different hardware for input/output. Therefore the set of mappings available on a machine may be restricted.

When Unger (1958) first proposed utilization of image parallelism he used a two-dimensionally connected array of processing elements and mapped only one pixel onto each element. This mapping has obvious advantages. Firstly, it utilizes parallelity to a maximal degree. Secondly, it is very simple since the arrangement of the processors is the same as the arrangement of the pixels. This makes the step from algorithm to program straightforward and uncomplicated. The control unit, broadcasting directives to the processing elements, can also be kept fairly simple.

However, not even with the largest arrays implemented can we count on having as many processing elements as pixels. A very common image size, 512 times 512 pixels, is sixteen times the size of the largest array implemented - the 128 times 128 array of MPP. This means that with the one-pixel-per-processing-element mapping, we are obliged to resort to dividing large pictures into smaller parts, something that often gives cumbersome rand effects. Furthermore, the mapping also results in very small directly accessible neighbourhoods.

Danielsson and Levialdi (1981) consider these drawbacks very severe, and they are critical to this image to array mapping. Given a two-dimensional configuration of the processors they advocate another

mapping - one subimage per processor.

For arrays with a very large number of processing elements a two- dimensional organization is very natural and probably the best. For smaller arrays, however, other configurations may be equally favourable. A linear organization, often combined with some other interconnection scheme, is one example. Having chosen a linear organization, the mapping of an image to the array can still be made in different ways.

5.3.2 Two-dimensionally organized arrays

Among processor arrays implemented and used for image processing there are three examples of two-dimensional organization, namely CLIP4, MPP and DAP. MPP and DAP have direct connection to four neighbours from each processing element, while the CLIP4 processing elements have direct connection to eight neighbours. Reported implementations of image processing operations on these machines (Fountain and Goetcherian,1980 (CLIP4), Batcher,1980 (MPP), Kushner et al.,1981 (MPP), Marks,1980 (DAP)) show different mappings of images to processor arrays. While the users of CLIP4 and MPP use the one-pixel-per-PE mapping, Marks, in his implementation on the 32 x 32 PE DAP, loads a 6 x 6 pixels subimage into the memory of each processing element when processing 192 x 192 pixels pictures.

Users of CLIP4 have no true possibility of choosing other mappings since the machine was built for one-pixel-per-PE. The input/output system is made for 96*96 pixels frames from TV pictures. Furthermore, the processing elements of CLIP4 are equipped with strictly combinatorial parts that allow a signal to flow through a series of elements during one clock cycle. With the very slow clock rate of CLIP4 this is an important feature. However, it loses sense if other storage methods than one-pixel-per-PE are used. Finally, the very small memories of the processing elements do not allow many pixels to be stored in one processing element.

DAP, on the other hand, was not built for image processing. No image input/output system is built into the machine. A signal is not allowed to pass through many PEs in a single clock cycle. Therefore, Marks as a user is free to adopt any storage method. The one he finds best is to store a square subimage in each PE, although it gives some problems with irregular addressing schemes.

MPP is not yet operable. With the fully synchronous processing elements and the powerful input/output reformatting hardware, use of other storage schemes than one-pixel-per-PE should be possible. However, Kushner et al.(1981) limit their analysis to 128*128 images, because they consider the memory of each PE too small for working on a larger subimage than 3*3 pixels. The reason for this is that they evidently store not only the subimage but also its immediate neighbourhood in the same memory. This seems to be unnecessary.

5.3.3 Linearly organized arrays

Like DAP, the STARAN computer (Batcher, 1974) was not primarily designed for image processing, but its use in this field of application has been thoroughly investigated (Goodyear,1976, Potter,1978). STARAN uses a linear ordering of the 256 processing elements of an array. In addition to this the so called flip network provides further possibilities for communication between processors as described in Chapter 1.

The image to processor array mappings used with STARAN follow roughly the approach one-pixel-line-per-processing element. When the number of lines exceeds 256, two or more lines are stored in each processor's memory as shown for a 512 x 512 pixels image in Figure 5.3.1. Lines stored in the same memory word will thus be 256 lines apart in the image. Two adjacent lines along the cut in the image will reside in the memories of the bottom and top processor respectively. If a "wrap-around" neighbour communication is used this should give no access problems. However, the addresses to neighbouring pixels will be different for these lines. This may very well double the

computation time for local operations.

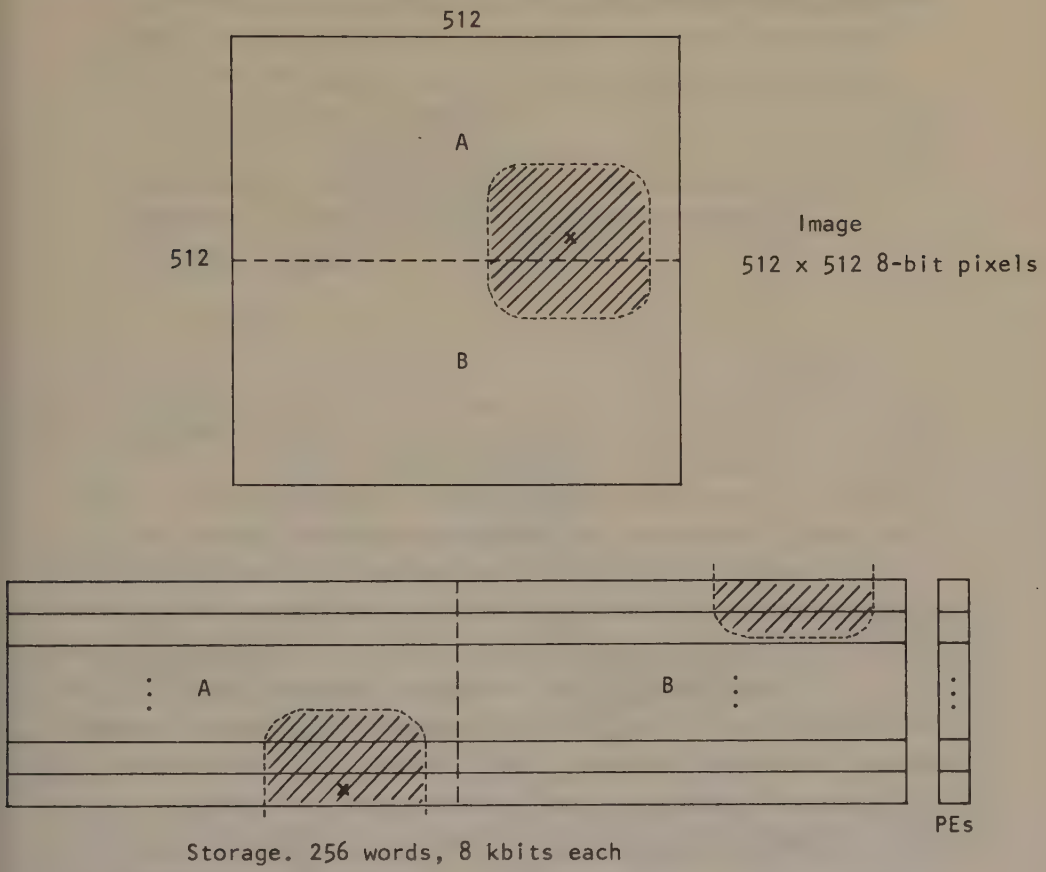


Figure 5.3.1 Storage of a 512 x 512 x 8 image in a 256 PE STARAN

Another approach to storing a 512 lines picture in a 256 processors array is to store two adjacent lines in the same memory word. The major advantage with this method is the larger immediately accessible neighbourhood that automatically follows. The method is illustrated in Figure 5.3.2. A 5 x 5 neighbourhood is directly accessible only with up/down communication. (In fact a 5 x 512 neighbourhood). Of course, the method can be generalized to any other ratio between image height and number of processing elements. The

larger the ratio, the larger will the size of the immediately accessible neighbourhood be.

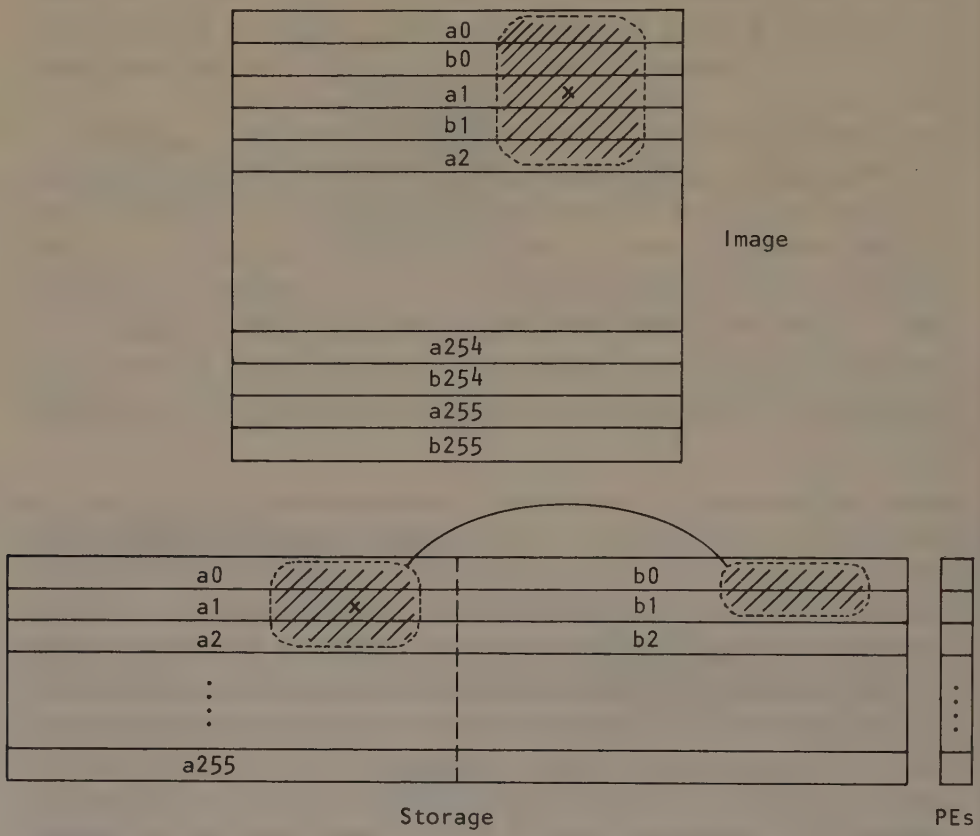


Figure 5.3.2 Alternative method for storing a 512 by 512 pixels image in a 256 PE array

When comparing these two different ways of image-to-storage mapping three aspects ought to be considered:

* The effect on input/output. The image normally comes in TV scan format. One storage method may be easier than the other to implement.

* The effect on computation time. The larger directly accessible neighbourhood in the second method should make computation of local operations faster. Also the "cut" in the image when using the first method may increase the computation time.

* The demands on the control unit. The computation of bit addresses to the memory words may be more elaborate in the second method.

5.3.4 Discussion

Which image subregion shape is the best when an image is subdivided between different processors is studied in (Kushner et al, 1980) with special reference to the ZMOB system. For local operations the optimal shape in order to minimize information passing between processors is found to be the square. However, in systems with very tightly coupled processors, fetching data from a neighbouring processor's memory may be as fast as fetching data from the processor's own memory. Thus, minimal information passing between neighbouring processors is not necessarily the main goal. Rather, one should aim at minimizing the information transfer between non-neighbouring processors.

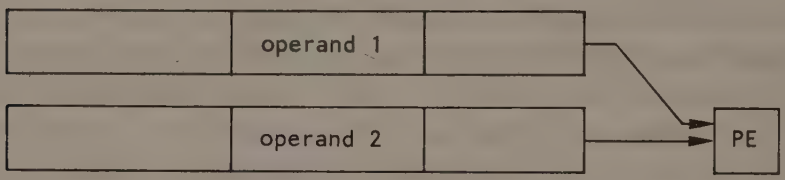
In (Swain et al.,1980) different image to storage mappings are compared. With M processing elements available to process an M*M picture, the two methods (a) one line per PE and (b) one $\sqrt{M}*\sqrt{M}$ subimage per PE are compared. The processing times for the specific task at hand (computation of the difference between the maximum and minimum elements of a 3*3 neighbourhood) are

$$T_a = 2M*A1 + (21M-24)*A2$$

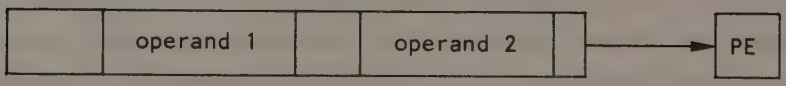
$$T_b = 6\sqrt{M}*A1 + 21M*A2,$$

where A1 is the extra time to transfer one data item from a PE to a neighbouring PE and A2 is the time for one operation. With a positive value of A1, the second method is superior to the first one. However, as noted above, fetching data from a neighbouring PE may take no extra time, meaning that A1=0. In that case the methods are (approximately) equal.

In a specific implementation it may even be the case that A1 is negative. If the arithmetic unit of a processing element has two separate input lines for arguments, an operation that involves data from neighbouring PEs may very well be faster than one that uses data from the PE's own memory only, see Figure 5.3.3. Thus, information transfer between neighbouring PEs may even be desirable in order to use the full bandwidth.



a) one read, one write per bit



b) two reads, one write per bit

Figure 5.3.3 An illustration of the fact that a two-argument operation can be faster if the two arguments reside in different memory modules (bit-serial PEs assumed)

In reported applications of PASM (Siegel et al.,1979) (pre-implementational studies) the method 'two-dimensional array' with 'one subimage per PE' is used for local operations. (E.g. a 16 by 16 part of a 512 by 512 image in each of 1024 PEs arranged as a 32 by 32 array). Since the PEs are ordinary microprocessors, there is a time penalty for transferring data between PEs. Therefore, in the light of the abovementioned analysis, this is probably the best arrangement.

For calculation of the two-dimensional discrete Fourier transform on PASM, another communication network between the processors is used, and also another image to storage mapping. This illustrates an important point: The input/output system of a processor array of this type should be flexible enough to provide possibilities for different mappings of an image to the total memory.

To conclude this discussion we note that, given a certain interconnection structure, the mapping of an image to this structure can still be made in different ways. Different mappings will result in different computation times, but also in different times for input/output. Depending on the computational task at hand, different mappings are optimal. The total time, input + computation + output, should be conclusive. For the choice of interconnection structure, the size of the array is probably important. A configuration suited for small arrays may suit badly for large arrays, and vice versa. The array of LUCAS is quite small. In the sequel we will only consider one configuration in order to be able to study in detail the execution of different image operations on such a structure.

5.4 IMAGE OPERATIONS ON LUCAS ORGANIZED AS A LINEAR ARRAY OF PROCESSING ELEMENTS

5.4.1 Introduction

In this section we will use LUCAS as a model machine in order to examine the applicability of bit-serial SIMD machines with up to a few hundred processors in the field of image processing . Algorithms are programmed and analyzed with regard to execution time and possible changes in the hardware that would make execution faster.

We do not claim any particular novelty for the algorithms presented, but rather our interest centers on the techniques of implementation and the level of performance achievable using this specific type of hardware.

The interprocessor communication structure of LUCAS is reconfigurable. Therefore different organizations can be tested. However, we will assume a linear organization of the processor array with communication one and two steps up and down in addition to a perfect shuffle/exchange network (Figure 5.4.1). Furthermore, we will assume that the size of the image side agrees with the number of processing elements so that one line of the image exactly occupies a field of the memory as shown in Figure 5.4.2. Only in the last subsection (5.4.8) will we depart from this assumption and discuss the consequences for neighbourhood size and input/output.

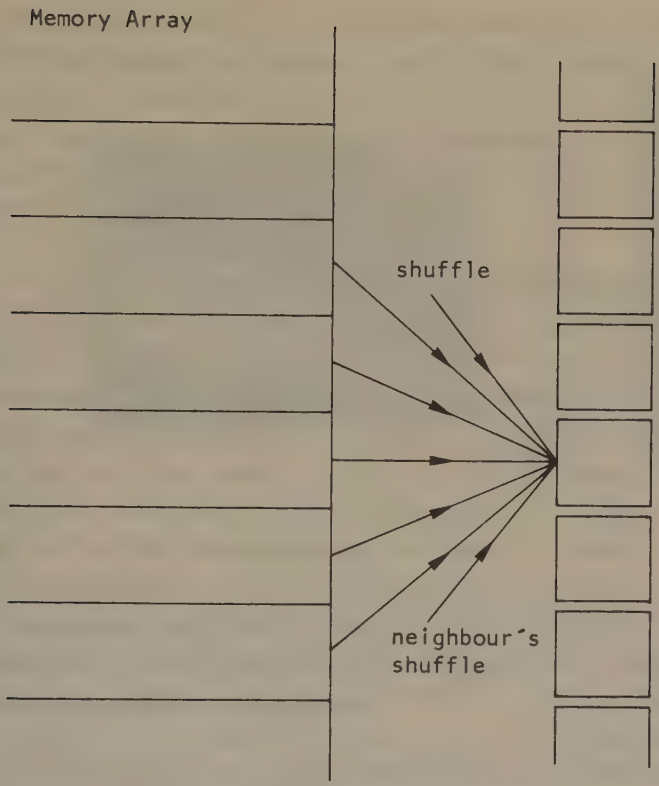


Figure 5.4.1 Available data inputs to a PE

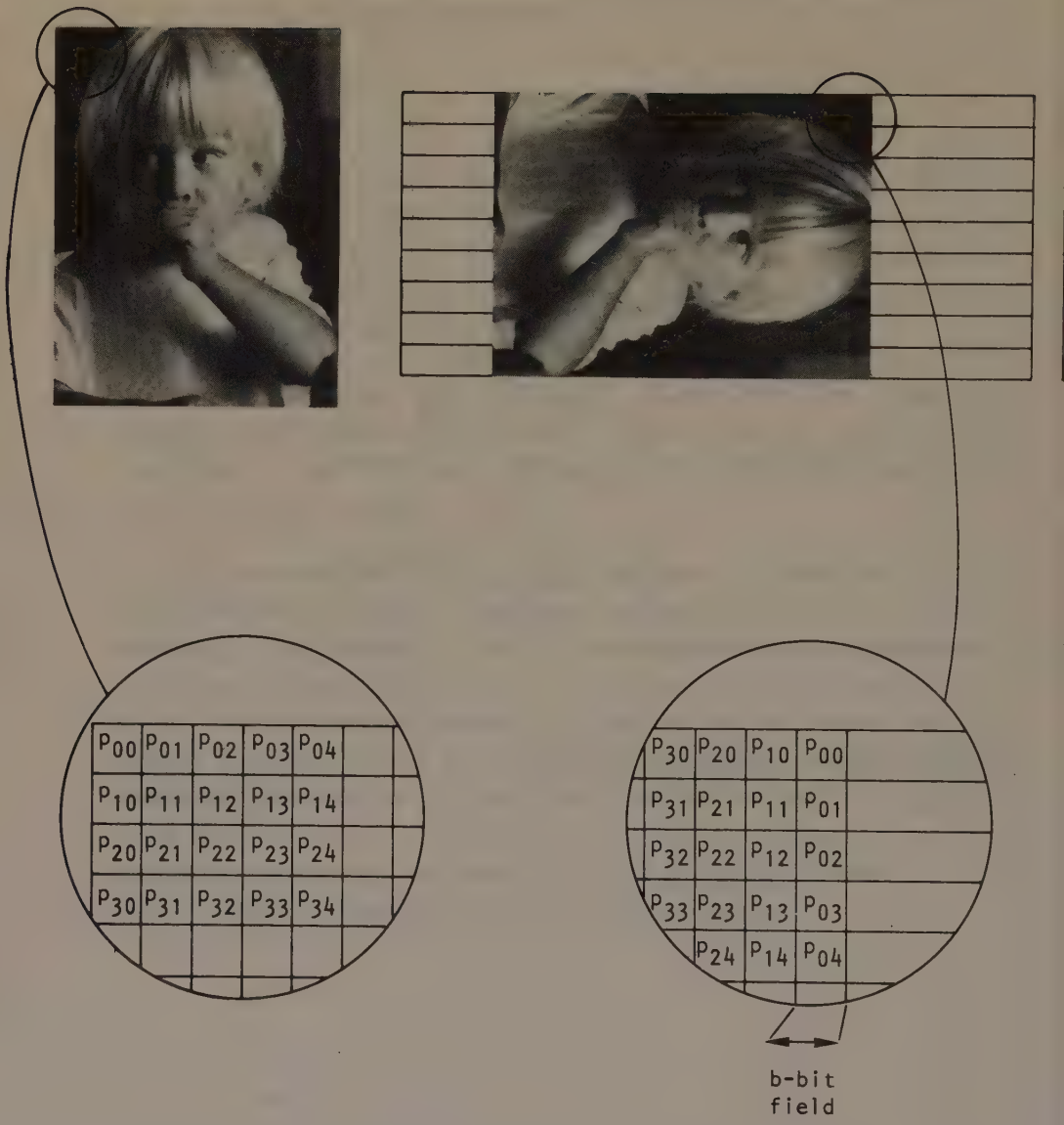


Figure 5.4.2 Storage of an image in the memory of LUCAS

The main concern in our investigation is the set of operations that takes images into images. Firstly, this is the kind of operations for which a processor array is best suited. Secondly, they are normally the most time consuming operations on ordinary computers. We will also briefly examine the use of LUCAS for extraction of picture properties of different kinds.

In the investigations to follow we will group operations according to the characteristics of their execution on LUCAS. In association with each operation treated, we will indicate in what context of image processing it is normally used. In most cases similar operations are described in (Rosenfeld and Kak, 1976) where more background material can be found. Unless otherwise stated, the presented algorithms have been microprogrammed and tested on LUCAS. A selection of the microprograms are given in detail in an appendix. Timings presented are made using a clock cycle of 200 ns. The microprograms are, with a few exceptions, general with regard to image width and pixel precision. These are specified as input parameters to the microprograms.

In connection with the presentations of the algorithms we will sometimes indicate changes in the hardware of LUCAS that would improve the performance.

5.4.2 Genuinely local operations. Small neighbourhood sizes

We call an operation genuinely local if the new value of a pixel (x,y), as the result of the operation, depends only on the pixel values in a small neighbourhood around (x,y). If O is an operator that consists of a sequence of such operations, O is no longer genuinely local, since the new value of a pixel at one side of the picture may very well depend on the old value of a pixel at the other side. This kind of operations are treated in a separate section below.

In this section we treat local operations with neighbourhood sizes smaller than or equal to 5 in one direction, arbitrarily large in the

other direction. (Usually, the neighbourhood is approximately quadratic). This means that all pixel values of the neighbourhood are immediately available over the interconnection network (Figure 5.4.1).

Binary images

EXAMPLE 1: Salt-and-pepper noise removal

A binary picture obtained from a grey scale picture by thresholding often has scattered white points in black regions and scattered black points in white regions as a result of noise in the original picture. This is called salt-and-pepper noise.

Salt-and-pepper noise can be detected by counting the number of neighbours of a pixel that differ from the value of the pixel itself. If this number is large, the value of the pixel is changed. An algorithm that changes a pixel value if it differs from seven or more of its eight nearest neighbours proceeds on LUCAS as follows.

The image is swept over pixelcolumn-wise, from right to left. Matches are counted instead of non-matches. A two bit field is reserved for a counter in each word. For each pixel column, the counter is first initiated to zero. Then, for each neighbouring point that matches, the counter is incremented. After the counter has reached binary 10, the most significant bit is locked to one. This bit will serve as a mark bit for noise points. Finally, the value of these points are changed and written into a separate result image, and the scan proceeds to the next pixel column.

The microprogram is included in full detail in the appendix. The execution time for an image width of w pixel columns is 71w + 7. An image with width 128 pixels is treated in 9095 clock cycles, i.e. 1.82 ms.

A few comments on the microprogram may be appropriate: A subroutine is used to add zero or one to the count field, depending

on non-match or match, respectively. This subroutine is called upon comparison of each of the eight neighbours. If there is no match, zeroes will be added to all counters, which is unnecessary.
Therefore, a test for some match could be included before calling the subroutine. However, a total non-match can only occur in very special types of pictures, like a set of vertical stripes over the entire picture, and is thus very unlikely.

More likely, in not too noisy pictures, is that, in a certain pixel column, no noise points are detected. In that case no changes need to be made. However, since in any case the result has to be written in the destination image, no time can be gained by testing for these cases.

When comparing each pixel in a column with the pixel below we get exactly the same result column as when we compare with the pixel above, only shifted one step up. (With reservation for the topmost and lowermost pixels). In the light of this, it may appear that unnecessary many comparisons are made. However, to move the contents of one of the flip-flops one step up or down takes longer time than to make an additional comparison. This is because the interconnection network always takes data from memory to processing elements, not from processing element to processing element.

A dramatic time gain would of course be achieved if a counter were included in each processing element. A counter that could be incremented and tested in one clock cycle would save 56 percent of the processing time.

For the task at hand we can also manage very well without actually counting the matches or mismatches. Instead of adding the mismatch indicator (1 or 0) to the counter field for each neighbour checked, we can just save the indicator in an 8-bit mismatch vector, which can be analyzed after the whole neighbourhood is gone through. If the mismatch vector contains two or more zeroes, the point is not considered a noise point. In order to find a microinstruction sequence that decides if the vector has two or more zeroes, we will use a state transition graph, hoping to be able to use the flip-flops

of the processing elements directly as state variables. (See Figure 5.4.3). Starting in state S1 and scanning the vector, we will go to state S2 upon finding the first zero. Finding one more zero will lead us to state S3 where we will stay.

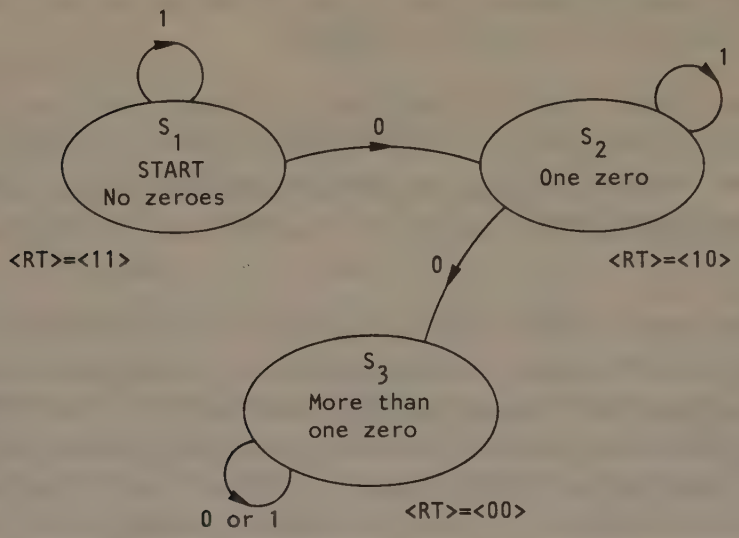


Figure 5.4.3 State transition graph for the mismatch checking

Using R and T as state variables and the coding shown in Figure 5.4.3, the next-state functions are

$$R^+ = T \vee (M \,\&\, R)$$

$$T^+ = M \,\&\, T, \text{ where M is the mismatch vector bit.}$$

These functions can be realized by the sequence

```
ANDRMA,D        ;M & R  -> R
ANDTMA,D        ;M & T  -> T   Old T to X
ORRMA,X         ;T v (M & R)  -> R
```

(All values to the left of the arrows are old values).

Thus the 8-bit mismatch vector is analyzed in 3 times 8, i.e. 24 clock cycles. Compared to the 48 cycles used for counting in the above solution, we have reached a significant improvement. The total time is decreased by almost 35 percent.

EXAMPLE 2: Border finding

A point in a binary image is called a border point if it has the value '1' and is adjacent to a point with the value '0'. Depending on whether we use 8-adjacency or 4-adjacency (see Figure 5.4.4) we get slightly different borders (Figure 5.4.5).

(a) (b)

Figure 5.4.4 a) the points 4-adjacent to x
b) the points 8-adjacent to x

(a) (b) (c)

Figure 5.4.5 a) object
b) border using 4-adjacency
c) border using 8-adjacency

A microprogram to mark border points is very straightforward. The image is swept over column-wise. For each column the logical product of the neighbourhood (4- or 8-) of each pixel is formed. If the product is zero and the pixel value is one the pixel is a border point. The logical product of a 3 times 3 pixels neighbourhood can be formed in only four AND-operations, two "horizontal" and two "vertical". Therefore, the 8-adjacency case gives only slightly longer execution time than the 4-adjacency case: 9w+6 and 8w+6, respectively. In time this means 0.23 and 0.21 ms, respectively, for w=128.

For a given pixel column there is always some probability, p, that there are no pixels at all with the value '1'. In that case the job to form the logical product of all neighbouring pixels is superfluous. If there is a cost involved in testing for this case, there will also be a certain probability, p_0, above which testing is rewarded with a time gain. For the border finding microprogram, testing will cost two cycles for each bitslice. When the test for some '1' comes out false, the saving is 6 cycles in the 4-adjacency case and 7 cycles in the 8-adjacency case. This gives

$p_0 = 2/6$ in the 4-adjacency case

$p_0 = 2/7$ in the 8-adjacency case.

If the probability for a slice to be empty is larger than this

value, there is gain in testing. Figure 5.4.6 illustrates the computing times graphically.

The appendix shows the version of the microprogram for the 8-adjacency case that contains tests.

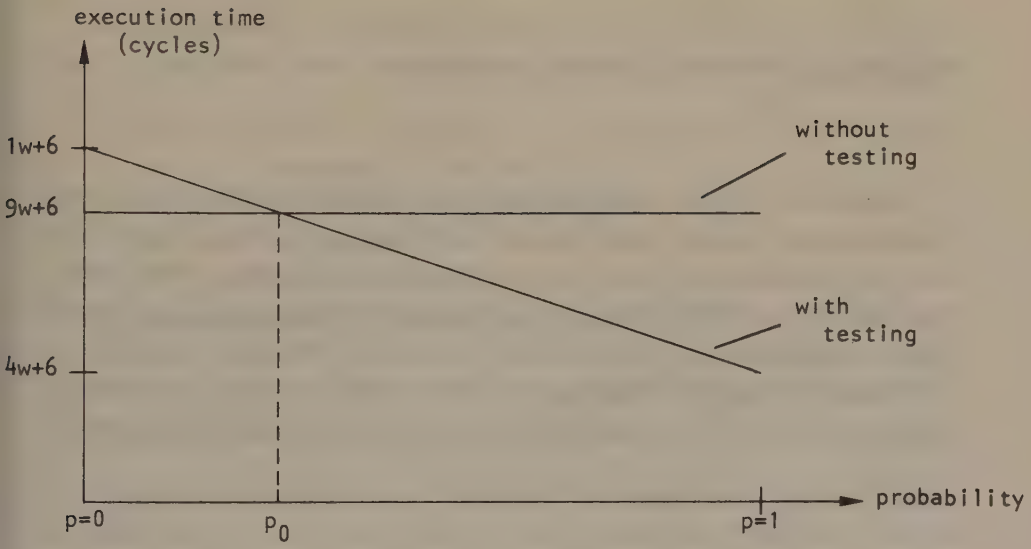


Figure 5.4.6 Execution times with and without testing (8-adjacency case)

EXAMPLE 3: Shrinking and expanding

The operations of shrinking and expanding in binary images have many applications (Rosenfeld and Kak, 1976). One or a few shrinks followed by the same number of expansions will clean up "ragged" borders and delete small objects, see Figure 5.4.7. Shrinking and

expanding can also be used to obtain the skeleton of an object or to detect clusters of points.

Figure 5.4.7 Shrinking followed by expansion

Shrinking is the same as deleting border points. Thus, the microprogram becomes very similar to the border finding program. The time for 4-adjacency shrink is 7w+6 and for 8-adjacency shrink 8w+6. The times are a little shorter than for border finding because, in the logical expression for the new value of a point, the point itself plays the same role as its neighbours. This is not the case for border finding. The times for expansion are the same as for shrinking.

EXAMPLE 4: Gap filling

As an example of an operation with a larger neighbourhood than 3 x 3 we use the following one (Iliffe, 1982), useful for filling in gaps in thin curves:

If some point in X and some point in Y have the value '1', let point 'Z' have the value '1'. (See Figure 5.4.8).

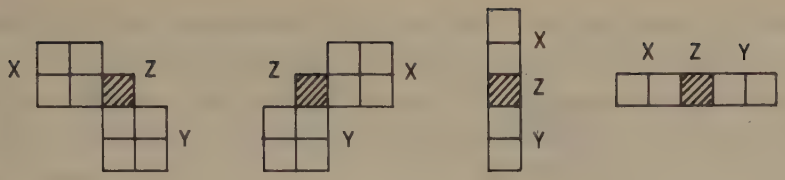


Figure 5.4.8 Mask configuration for gap filling

A straightforward approach to solve this on LUCAS is to take the four cases one after the other, OR-ing the results together:

Start by clearing a scratch pad bit-slice SP. For each case:

```
OR-sum of X-field -> R -> T
OR-sum of Y-field -> R
R AND T -> R
R OR SP -> SP
```

Then: SP OR Z -> Result image

The time for the execution of the microprogram is 45w+6 cycles. w=128 gives 1.15 ms.

There are possibilities to improve the algorithm. E.g. the OR-sum of a certain 2 x 2 pixels area is used to calculate the new value of no less than 4 pixels. See Figure 5.4.9. For calculation of each of the black points the OR-sum of the white points are needed. Therefore it would be a good idea to first create the OR-sums of all 2 x 2 areas. This, in turn, is best done by first creating the OR-sums of all 2 x 1 or 1 x 2 areas.

Figure 5.4.9 For calculation of the values of each of the black points the OR-sums of the white points are needed

Grey scale images

Many local operations on grey scale images include additions and subtractions of whole images. Often, the two images in such an operation are identical except for one of them being shifted one step. This is the case in averaging and differentiating operations, to name two examples. Therefore, our first examples will be these, generally useful, operations. Later examples will combine these to compound operations.

EXAMPLE 5: Addition/subtraction of images

Addition (or subtraction) of two pictures takes three cycles per bitslice and four additional cycles for each pixel column (for test and reloading of bit counter). Including initial parameter loading this makes

w(3b+4)+6 cycles,

where w is the image width in pixels and b is the number of bits per pixel, both specified through parameters. (The additional 4 cycles per pixel column can be reduced to one if the operation is made as a single w*b bits wide addition with markers in the mask register for pixel-slice limits).

Addition of two 128 pixel wide images with 8 bit data takes 3590 clock cycles, i.e. 0.72 ms.

EXAMPLE 6: Point by point maximum of images

A point by point maximum operation on two images A and B replaces all pixels of A, that are smaller than the corresponding pixels of B, with the B-pixels. The operation proceeds in two phases: In the first phase a pixel slice of B is subtracted from the corresponding pixel slice of A without storing the result. The signs of the result is moved to the tag flip-flops. In the second phase the B-slice is written tag-masked into the A-image slice. The time for subtraction is two cycles per bitslice (two reads). The time for move is also two cycles per bit-slice (one read, one write). The total time is

w(4b+10)+4 cycles.

w=128 and b=8 gives 5380 cycles, or 1.08 ms.

EXAMPLE 7: Thresholding

The most common operation used for segmentation of grey scale images is thresholding. It produces a binary image that has ones in those coordinates of the original picture where the value exceeds a certain threshold. In the implementation on LUCAS, two cycles are used per bit slice. This is because the threshold value is stored in

each memory word. An alternative is to store the threshold in the Common Register. Since the ALU has one input from Common and one from the memory word, the comparison could then be made faster. However, the Common Register receives the same address signals as the memory. Hence the threshold must be stored repeatedly along the Common Register. A base register for Common Register addressing would be a good thing to include in the Control Unit.

The implementation of thresholding made on LUCAS takes

$$w(2b+5)+7 \text{ cycles.}$$

w=128 and b=8 gives 2695 cycles, or 0.54 ms.

EXAMPLE 8: Roberts' cross-difference operator

Difference operators are widely used for the detection of edges. One variation of the so called Roberts' operator (Roberts, 1965) has the following form (j and k are image coordinates):

$$R(j,k) = \max(|I(j,k)-I(j+1,k+1)| \ , \ |I(j,k+1)-I(j+1,k)|)$$

The microprogram for this can be divided into two subtractions, two formations of absolutes, and one maximum. Subtraction and maximum were treated above. Absolute value formation on a 128 x 128 8-bit image takes 2820 cycles, or 0.56 ms. Thus, the execution time for Roberts' cross-difference operator is

2 subtractions:	2 x 0.72 ms
2 absolutes:	2 x 0.56 ms
1 maximum:	1 x 1.08 ms
Total execution time:	3.64 ms

An example showing the effect of the operator is given in Photo 5.4.4 under example 16 below.

EXAMPLE 9: The Laplacian operator

The derivative of an image in the x-direction can be approximated by the expression

$$df/dx = f(x+1,y) - f(x,y)$$

The second order derivative, then, becomes

$$d^2f/dx^2 = [f(x+1,y) - f(x,y)] - [f(x,y) - f(x-1,y)] =$$
$$= f(x+1,y) + f(x-1,y) - 2f(x,y)$$

The Laplacian operator is defined as

$$L(f) = d^2f/dx^2 + d^2f/dy^2 = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

The application of the Laplacian to a pixel whose four neighbours all have the same value as the pixel itself, gives a zero result. If some of the neighbours have smaller values but none has larger, L(f) will be negative. This is the case, for example, at one side of an edge in the image, At the other side of the edge, some values have a larger value than the center pixel, but none has smaller. There the result will be positive.

The function

$$f(x,y) - L(f(x,y)) = 5f(x,y) - [f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)]$$

will take on the value f(x,y) at all points where the mean value of the 4-pixel neighbourhood is the same as the center pixel value. It will take on a value smaller than f(x,y) if f(x,y) is smaller than that mean

value, and it will take on a larger value if f(x,y) is larger than the mean value of the neighbourhood. Thus, application of this function has the effect of increasing the contrast in the image.

The Laplacian operator applied in the way described above can be used for enhancement of blurred pictures and also for detection of edges, lines and spots. Alternative digital "Laplacians" can be defined by using different neighbourhoods, or by using a weighted average over the neighbourhood.

We will consider the implementation on LUCAS of two operations of this kind, L_4 and L_8. L_4 is the inverse of L as defined above. L_8 uses all pixel values of a 3x3 neighbourhood. L_4 and L_8 are linear filters that can be described as in Figure 5.4.10.

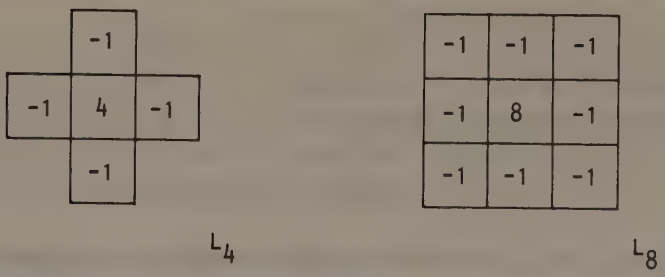


Figure 5.4.10 The operators L_4 and L_8

Photo 5.4.1 illustrates the effect of using L_4 on an image. The upper left image shows the original image, reproduced with only 4-bit grey scale. Upper right is the result of applying L_4. Negative values are put to zero. Addition of the result to the original image gives the result shown lower left. The contrast has increased compared to the original image. Subtraction instead of addition gives the result shown lower right. Here the edges have been blurred.

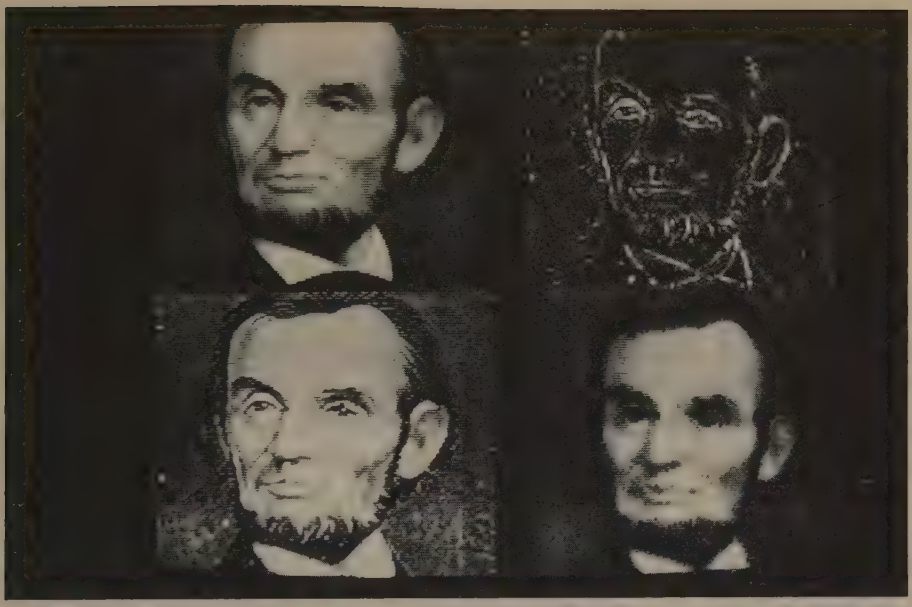


Photo 5.4.1 Illustration of the effect of the Laplacian operator L_4. Upper left: original. Upper right: L_4 applied. Lower left: L_4 added to original. Lower right: L_4 subtracted from the original. Negative values have been set to zero. 4-bit grey scale is used.

Computation of L_4 is done by first multiplying all pixel values by the factor 4, then subtracting the values of the four neighbouring pixels, one after the other. Multiplication by 4 takes no time at all, since it only means changing the address when bits of the pixel are fetched. Thus, the total time is the time of four subtractions. We can reduce this to the time of two additions and one subtraction with the following method: First, the sum of each pixel and its upper right neighbour is formed and stored in a temporary area. Then, each pixel in this area is added to its upper left neighbour. The result obtained is finally subtracted from the original image value immediately to the right multiplied by 4.

If the full dynamics of the b-bit fields is to be used, extension of the field length to b+2 bits must be made. Thus, the time to compute L_4 is the following (time for addition and subtraction was given in

example 5 above):

$$T(L_4) = 3*[w(3(b+2)+4)+6] = w(9b+30)+18 \text{ cycles.}$$

w=128 and b=8 gives $T(L_4)$ = 13074 clock cycles, i.e. 2.61 ms.

To compute L_8, the sum of all nine elements of the neighbourhood is first computed, which can be done in just four additions. First, the neighbour above is added to each element, then the neighbour below in the original image is added to this sum. In the new image obtained in this way the process is then repeated, using the left and right neighbours, respectively, instead of those above and below.

To give L_8, the sum of all nine neighbourhood pixels is to be subtracted from the value of the center pixel multiplied by 9. Multiplication by 9 is performed as an addition of the pixel value with itself shifted left three positions. The total sum for the computation of L_8 becomes the sum of:

a) Vertical addition: 1 addition using b bits, 1 using b+1 bits. b) Horizontal addition: 1 addition using b+2 bits, one using b+3 bits
c) Multiplication by 9: 1 addition using b+3 bits
d) Subtraction: 1 subtraction using b+3 bits

The computation times are:

a) w(6b+11)+12
b) w(6b+23)+12
c) w(3b+13)+12
d) w(3b+13)+12

The total time is

$$T(L_8) = w(18b+60)+48 \text{ cycles.}$$

w=128 and b=8 gives $T(L_8)$ = 26160 clock cycles, i.e. 5.23 ms.

EXAMPLE 10: Mean value filtering

In some pictures, replacing the value of each point with the average pixel value in some neighbourhood of the point (including the point itself) may be a useful way to reduce noise. This is called local averaging or mean value filtering.

Mean value filtering means addition of all pixels in the neighbourhood followed by a division by the size of the neighbourhood. We consider mean value filtering over neighbourhoods of size 3x3 and 5x5. Division by 9 can be approximated by a multiplication by 7/64 with an error of about 1.5% only. (Anyhow, with limited data length, division by 9 cannot be done exactly). Similarly, multiplication by 5/128 is an approximation of division by 25 with an error of 2.3%. Multiplication of a value by 7 can be implemented as a multiplication by 8 (which takes no time) followed by a subtraction of the original value. Division by 64 takes no time, which means that division by 9 can in fact be done as a single subtraction. Similarly, division by 5 will be a single addition.

The sum of all pixels in a 3x3 neighbourhood is obtained through four additions, as described in example 10 above. The time for this is w(12b+34)+24. The subsequent division by 9, which is realized as a single subtraction, takes w(3b+13)+6 cycles if truncation to b bits is postponed till after the division. In total, the time to compute the mean value of each 3x3 neighbourhood in an image is

$$T(M_9) = w(15b+47)+30 \text{ cycles.}$$

w=128 and b=8 gives $T(M_9)$ = 21406 clock cycles, i.e. 4.28 ms.

In the case of a 5x5 neighbourhood, the sum of all elements in the neighbourhood can be obtained in six additions - three vertical and

three horizontal. The three horizontal additions can be reduced to two using the fact that the image is swept over columnwise from right to left during computation. The sum over the neighbourhood of a specific pixel can be computed from the one obtained for a pixel in the preceding column by subtracting the rightmost contribution and adding a new contribution from the left.

The time to compute the sum of the neighbourhood for each pixel of the image is w(15b+62)+30 cycles. The final multiplication by 5 is done as a single addition on b+5 bits data. This takes w(3b+20)+6 cycles. Thus, the total time required to compute the average over 5x5 neighbourhoods is

$$T(M_{25}) = w(18b+82)+36 \text{ cycles.}$$

w=128 and b=8 gives $T(M_{25})$ = 28964 clock cycles, i.e 5.79 ms.

The time to do averaging over a 5x5 neighbourhood is only 35% longer than the time required for a 3x3 neighbourhood.

EXAMPLE 11: Median filtering

For suppression of noise in images, the use of non-linear filters, like the median filter, is considered to have many advantages over linear filters, e.g. taking the average. Perhaps the most important is the ability to preserve sharp edges (Justusson, 1980).

Median filtering means replacing a pixel value by the median of the neighbourhood. Danielsson (1981) has devised an algorithm that utilizes bit-serial scanning of the arguments. The algorithm has been implemented on LUCAS for a 3*3 neighbourhood. It starts by analyzing the set of most significant bits of the neighbourhood points. If there are more zeroes than ones in this set, it can be concluded that the median value has a zero as its most significant bit. It proceeds with the following bits, successively refining the hypothesis.

When traversing the neighbourhood, scanning a bit slice of the arguments, certain conditions have the effect that a counter of each point is incremented, while certain other conditions have the effect that the counter is decremented. Since this operation on the counter is done bit-serially on LUCAS, it takes a considerable part of the total execution time - around 70 percent. The execution time for a w-column picture with b-bit pixel values is

$$w(154+324b) \text{ cycles.}$$

Our example w=128, b=8 yields 351,488 cycles, i.e 70 ms. If the processing elements were provided with counters that could be incremented or decremented in one cycle, the time would decrease to around 20 ms.

5.4.3 Genuinely local operations. Larger neighbourhood sizes

The communication network on LUCAS permits a processing element to access data from words one or two steps up or down. When data is needed from words at a larger distance from the PE, it must be temporarily loaded in words in between. The larger the distance is, the larger number of temporary storage steps are needed. When the distance is very large - 20 or more, approximately - it may be favourable to use the perfect shuffle/exchange network to route data to the desired PEs. Shift of data an arbitrary number of steps up or down can be made in $\log_2 N$ passes through the network, where N is the number of PEs (Lawrie, 1975). Such long distances hardly occur in local operations.

In this section we will give one example of a local operation that uses a larger neighbourhood than is directly accessible. The example given concerns filtering of a grey-scale image; according to Kruse (1977) the need for larger neighbourhoods is stronger on grey-scale images than on binary images. The calculations involve multiplications, therefore the computation time will dominate strongly over the time to

fetch data to the correct PEs.

EXAMPLE 12: Linear filtering

The mean value filter described in example 10 above is an example of a convolution of the image matrix with a smaller matrix, in that case a 3x3 or 5x5 matrix with all values equal to 1. The convolution was followed by a division by the total weight of the convolution matrix in order to keep the overall grey-scale level in the image unchanged.

Linear filters are often specified as convolution matrices of larger size than this. Cross correlating the image with small template images also involves the same computations.

As our example we will take a linear filter specified by a convolution matrix of size 9x9. b bit data are used, both for image pixel values and filter constants.

The convolution is computed as iterations over (a) the pixel-columns of the image and (b) the 81 values of the convolution matrix. Depending on which one is chosen as the outer loop variable, two different computation strategies are obtained:

(1): For each of the 81 values of the convolution matrix, do the following: Multiply the entire image by the value. The product obtained in a specific point, p, is to contribute to the result of another point, located at a certain distance from p. This distance corresponds to where the currently used filter constant is located in the convolution matrix. In 36 of the 81 cases (see Figure 5.4.11) the transfer of the product must be done in two steps because of the limitations of the interconnection network. However, the overhead from this can be reduced if the products are stored closer to the final destination already when they are formed. In fact, this reduces the overhead to one clock cycle for each bit to be transferred.

(2): For each of the pixel columns: Multiply the column and its 8 neighbouring columns (4 on each side) by 9 values each. (In total 81 multiplications). After each multiplication, transfer the result to the PE that needs it and accumulate it. The overhead due to the limitations in interconnections is the same as above.

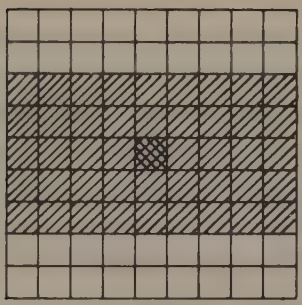


Figure 5.4.11 The area whose values are needed to compute the result at the center position. The shaded area shows which values are directly accessible

Measuring only pure computation time, the two approaches are equivalent. An advantage of method (2) is that computation can start as soon as a few columns of the image are input, and output can start as soon as the result of one column is obtained. Thus, computation and input/output can be overlapped. In the following, we will use method (1), but the timing for the other method will be the same, disregarding overhead.

In each of the 81 iterations, the entire image is multiplied by a scalar. Preferrably, the scalar is in canonical signed digit code (see Section 2.4.2). The time for the multiplication is, on the average,

equal to $b^2 + 4b - 3$ cycles. The products obtained are accumulated using an increasing number of bits, on the average 2b+5 bits. This takes 3(2b+5)+4 = 6b+19 cycles. Thus, the treatment of a pixel slice takes

$$b^2 + 10b + 16 \text{ cycles per iteration.}$$

This is done for all w columns in each of 81 iterations. The total time, then, becomes

$$81w(b^2+10b+16) \text{ cycles.}$$

With the values w=128 and b=8 this makes 1,658,880 cycles. To this value we should add the extra time required to pass data between the memory and the PEs due to the limitations of the interconnection network. This is the case in 36 of the 81 iterations, and the extra time is one clock cycle per bit to be transferred. This makes a total of 36 x w x 2b cycles, which for w=128 and b=8 is 73728 cycles. This is small compared to the computation time.

The total execution time for an 8-bit 9x9 linear filter on a 128x128 image of 8-bit data amounts to 1,732,608 cycles, on the average, i.e 0.42 seconds. The uncertainty of this value is large, since this algorithm has not been programmed on LUCAS.

5.4.4 Semi-local operations

Operations consisting of repeated applications of local operations until some specific criterion is reached form an important class of image processing algorithms. Although such operations are made up of local operations they can not be called genuinely local, since the result at some point in the image may depend on pixel values at a large distance from the point. We use the term "semi-local operations" to describe these operations.

Semi-local operations can be used for such tasks as counting the number of objects in an image, labeling objects or following curves

and borders. We will take four examples, all on binary images.

EXAMPLE 13: Connectivity preserving shrinking to a point

A method for counting the connected components (the objects) of a binary image is to shrink every object to a point and then count the number of '1's in the image.

We assume that the 4-adjacency relationship is used to define connectedness. We further assume that the components are without holes, otherwise the algorithm that we present here will not shrink them to a point. (There are algorithms (Rao et al.,1976) that also shrink objects with holes to single points).

The shrinking process is not allowed to disconnect any object. Therefore, the simple shrinking operator used in example 3 cannot be used. It would delete thin parts of the objects and thereby disconnect them. Instead, the operators shown in Figure 5.4.12, found in (Danielsson, 1982), will be used. The operators change the center pixel (underlined) from 1 to 0 if the neighbourhood is as specified. Repeated application of the operators will shrink objects to single points.

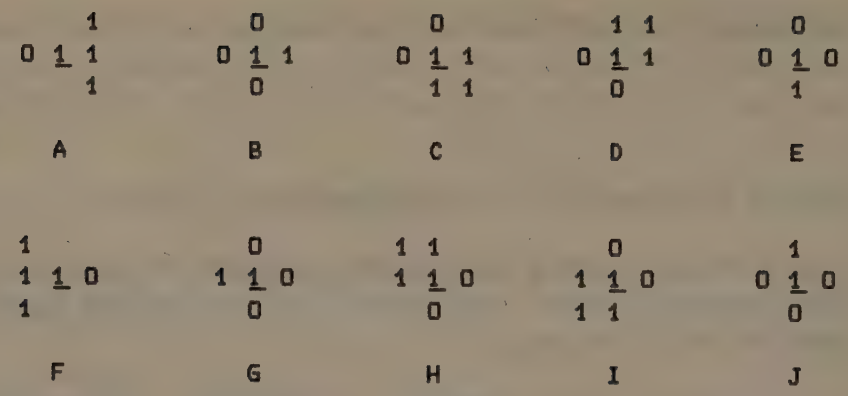


Figure 5.4.12 Connectivity preserving shrinking operators

As usual, the image will be swept over columnwise. All the operators are applied to a column before stepping to the next one. We will use what is often called "sequential" or "recursive" operating mode, meaning that the very input image is changed as the result of an operator. Thus, when applying the next operator on the same slice, the slice may have changed. This also motivates repeated application of the same operator on a slice before taking the next operator.

The algorithm that we use works as follows:

Scan the image from left to right.
For each pixel column:

1) Apply operator A.
2) Apply operators B, C and D in sequence.
3) Apply operator E and repeat it until no more changes occur.
4) Apply operator J and repeat it until no more changes occur.
5) Repeat steps 2,3 and 4 until no more changes occur.

Then, scan the image from right to left.
For each pixel column:

6) Apply operator F.
7) Apply operators G, H and I in sequence.
8) Apply operator J and repeat it until no more changes occur.
9) Apply operator E and repeat it until no more changes occur.
10) Repeat steps 7, 8 and 9 until no more changes occur.

Repeat the scanning procedures until a whole scan is made without any changes.

Before a new pixel column is treated a test to see if there are any '1's at all is performed. If there are no '1's in the column, applying the operators is a waste of time. Also, with the application of the operators A, B, F or G, a column may become blank. Therefore, after each of these operators, the test is performed.

The order between the individual operators is crucial. The operators A and F have the potential to delete a whole string of '1's in one application. Therefore, they are used first on each column. Once applied on a column, there is no sense in applying them once more, since no result from the other operators can make them applicable on new pixels. However, there are cases when another order would have been better. For example, when all objects are horizontal bars, one pixel wide, it would have been optimal to start with B and G.

The execution time for the procedure strongly depends on the characteristics of the objects in the image. On some images, one pass over the image is sufficient to delete all pixels but one per object. Normally, however, two passes or more are needed. Objects formed like spirals are the most difficult to shrink and require many passes.

An example where three passes are needed is shown in Figure 5.4.13. Pixels deleted in the first pass are marked by a '1', Those deleted in the second pass by a '2', etc. In the fourth pass no deletions are made. When this is discovered the procedure ends.

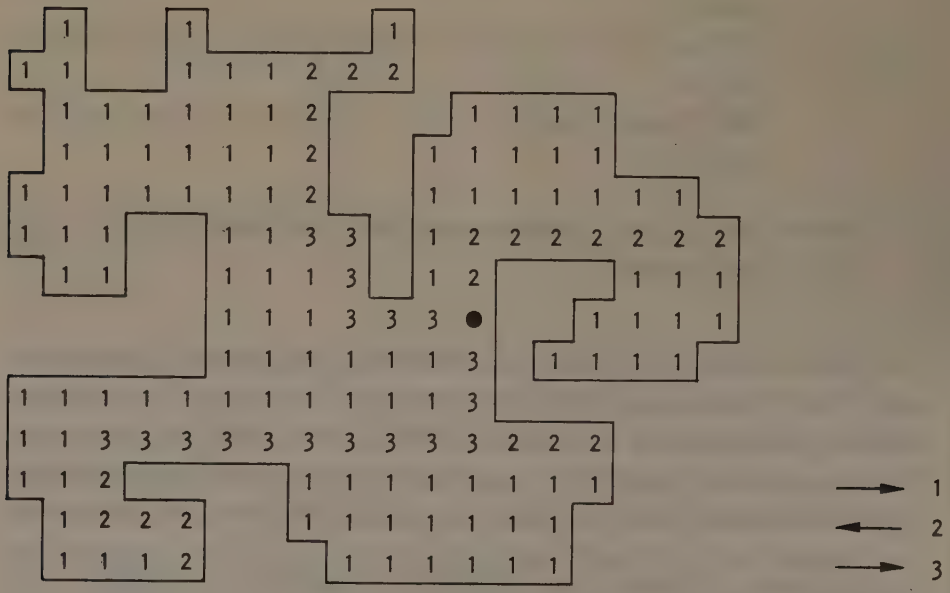


Figure 5.4.13 An object that is shrunk to a point in three passes

Photos 5.4.2 (a) – (c) show the shrinking of 13 objects in a 128 x 128 binary picture. (a) shows the original image, (b) shows the image after one sweep from left to right and (c) shows the final 13 points. The total execution time is 10 ms.

(a)

(b)

(c)

Photo 5.4.2 Connectivity preserving shrinking to points. (a) original image, (b) after one sweep from left to right, (c) final result

EXAMPLE 14: Finding the outer perimeters of objects

A method for finding the outer perimeter of each connected component (object) in a binary image is to propagate a marker from a point at the edge of the picture over the image area, until they reach an object. When this procedure is completed, those pixels of the objects that have markers as neighbours are marked as outer perimeter pixels.

As a matter of fact, the procedure can equally well be used for finding the holes or inner contours of objects. Hole points are those "background" pixels that have not been marked, and inner contours are those object pixels that have a hole point as a neighbour.

Different strategies can be used in order to spread the marker over the background as fast as possible. Figure 5.4.14 illustrates two approaches. We assume that 4-adjacency is used to define connectedness of objects. In both strategies the image is scanned back and forth. For each column, a pixel is marked if it has any 8-neighbour that has been marked. In (a) a new column is taken all the time, whereas in (b) a column is not left until no more pixels can be marked. The starting point at the edge is marked by a *. The numbers at the image points show in which step of the procedure a particular pixel is marked. In (a) the last pixel is marked in step no. 32, in (b) all pixels are marked after 15 steps. Furthermore, many of the steps in (b) can be shortened: when acting on a certain bit-slice, the horizontal and diagonal neighbours need be considered only in the first step. During the following steps, only the neighbours below and above can affect the result. Thus, it seems that the strategy that spreads the marker vertically to a maximal degree before continuing in the horizontal direction is the best one.

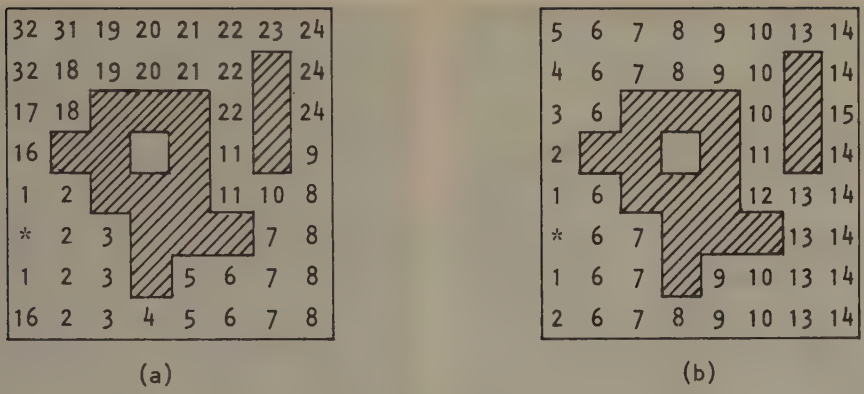


Figure 5.4.14 Different strategies for propagation of marker

It can be noted that propagations like this are very efficiently performed on the CLIP4 processor. The reason is that CLIP4 is equipped with a propagation function that is entirely combinatorial. Thus, the entire propagation is achieved by a single instruction. On LUCAS, we could imagine a similar function, working only in one dimension - vertical. It would be easy to implement.

Strictly synchronous twodimensional arrays (as to our knowledge, MPP is such) that store one pixel per PE will of course perform well on this operation, however not as many times better than a linear array as could be expected. On the example of Figure 5.4.14 a twodimensional array of 64 processors would need 9 steps to reach the last pixel, compared to 15 steps needed by a linear array of 8 processors. Each step requires looking at all eight neighbours. Larger examples that we have studied show that this tendency holds - the increase in performance falls far below the increase in amount of hardware.

A microprogram on LUCAS for finding the outer perimeters of objects is included in the appendix. Photo 5.4.3 shows an example of its use. The processing of the 128 x 128 image shown took 1.4 ms.

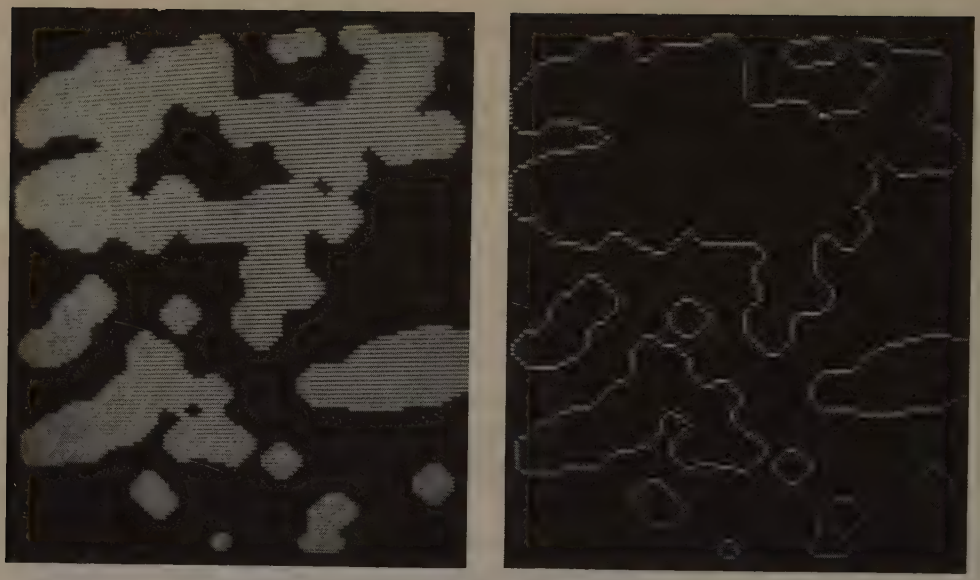


Photo 5.4.3 Finding the outer perimeter of objects

EXAMPLE 15: Component labeling

Component labeling in binary picture is the process of assigning different labels to different components of the image; in other words, for any component C, we want all points of C to have the same value, and no point not in C to have that value. The best method for doing this on LUCAS is probably to start off with a connectivity preserving shrinking operation, as suggested by Danielsson and Ericsson (1982). After the shrinking process, the image is scanned once more. For each one-valued pixel a new label is stored in the result image - the labeled image - at the corresponding position. Now, the labels of the

points are propagated to all points belonging to the same object in the original image. This is a process similar to the propagation described in the previous example. However, in this case not only a marker bit is propagated, but also - in a different memory area - a multi-bit label.

As with the previous examples, the processing time strongly depends on the shape of the objects in the image. The operation has not been programmed on LUCAS. A qualified guess is that the time for propagating labels is approximately the same as for connectivity preserving shrinking, assuming at most 64 different labels.

EXAMPLE 16: Tracking

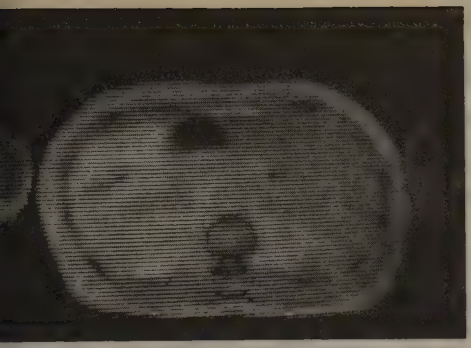
For the detection of edges in an image some kind of gradient operator (e.g. Roberts' cross difference operator, described in example 8) is often applied. (Possibly, some kind of preprocessing, e.g. median filtering, is first done in order to suppress the influence of noise). The derived picture is then typically thresholded at some appropriate lavel. A too high level will lead to some edge points being missed, while a too low threshold will give many "false" edges. A method that can be used to remove these drawbacks is "tracking". We then start with the "safe" edge points obtained by thresholding with a high threshold value (see image A in Figure 5.4.15). Then we propagate these points along connected edges in a picture that has been obtained by thresholding at a lower level (image B) and obtain an image of true edges (image C).

Figure 5.4.15 Result of thresholding at high level (A) and at low level (B). Result of tracking the '1's of A in B is shown in C.

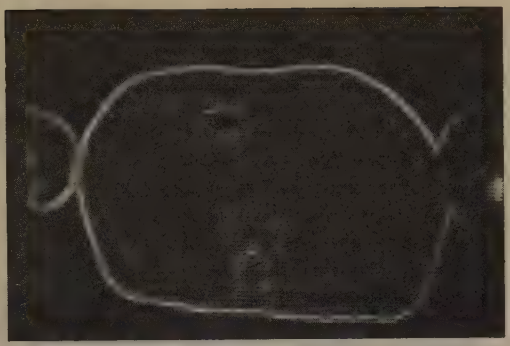
The technique used for propagation is exactly the same as in example 15. An example is shown in Photo 5.4.4 (a) ... (e). (a) shows the original image, (b) shows the result of Roberts' gradient operator (see example 8) applied to the image, (c) and (d) show the result of thresholding (b) using two different levels. (e) finally, shows the result obtained when the points in (c) are tracked along the edges of (d). The processing times to reach the different results are

b) 3.64 ms (Roberts')
c) 0.54 ms (Threshold)
d) 0.54 ms (Threshold)
e) 0.77 ms (Tracking)

The image size is 128 x 128 8-bit pixels. However, in the illustration in Photo 5.4.4 (a) and (b) only 4-bit grey scale is used.

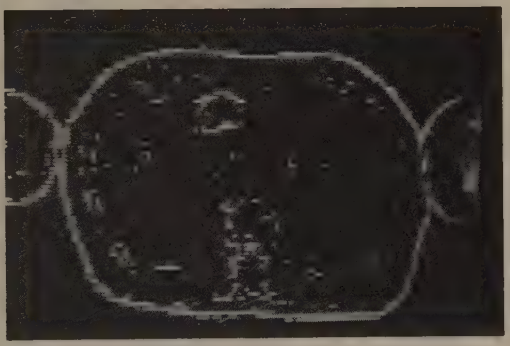


a



b

c



d

e

Photo 5.4.4 (a) original, (b) result of Roberts' cross difference operator applied to (a), (c) result of thresholding (b) at level 10, (d) result of thresholding (b) at level 4, (e) result of tracking the points in (c) along the points in (d).

5.4.5 Measurements

The operations that we have looked at so far have all been of the kind that transforms images to images. Often, we instead want to measure things in the image, e.g. count the number of objects, determine the area of the perimeter of an object, etc. This is also known as feature extraction.

Looking closer at such measurements, one finds that many rely upon a count of the number of '1's in a binary image. It is self-evident that those mentioned above do. Another measurement that is sometimes useful for pattern analysis is the following: Apply a shrinking operator to a binary picture repeatedly. After each step, count the number of remaining '1's. The successive counts will form a "feature vector" that will have quite different characteristics if the objects are small or large, elongated or not, etc.

As a matter of fact, a quantitative measure of the "elongatedness" of an object can be obtained through a study of shrinking. Let t be the number of shrinking steps required in order to erase the object totally. This means that the width of the object is 2t. Now, if the object has a quadratic form, the area is $4t^2$. Let A be the true area of the object. The quotient $A/(4t^2)$ is then a measure of the elongatedness of the object. Thus, we first measure the area of the object, i.e. we count the number of '1's in the binary image. Then we shrink the object until it vanishes and count the number of shrinking steps required. These two measures can then be used to get a value of the elongatedness of the object.

Counting the number of '1's in a binary picture is one example that we will consider in this section. Another is finding the maximum pixel value of an image, together with the coordinates of that pixel. The third example that we will treat is histogram collection.

EXAMPLE 17: Counting the number of ones

We will discuss two methods for counting the number of '1's in a binary picture. The second method assumes additional hardware, not yet implemented with LUCAS.

Method 1.

The first step in this method is the summing of each row of the image separately. This is of course done in parallel for all rows (words). The fastest way is the following (assume a 128x128 binary picture): First, sum pixels pairwise so that 64 sums, each with a value between 0 and 2, are formed. Then sum these sums pairwise, giving 32 sums with values between 0 and 4, etc. The reason for this method being efficient is the circumstance that the initial additions use very few bits, although the additions are many, and that the longer additions towards the end of the procedure are very few. In total, a little less than one thousand clock cycles are needed to sum over the rows.

Now, the row sums can be added fastly over the perfect shuffle/exchange network. Seven addition steps are required to add all 128 row sums. The number of bits increases from 8 to 14 during the process, which requires approximately 300 clock cycles.

In total, then, 1300 clock cycles are needed to count the number of '1's in a binary 128x128 picture. With a cycle time of 200 ns, this takes 260 microseconds.

Method 2.

If LUCAS is equipped with special purpose hardware to count the number of responders (number of Tag flip-flops equal to one), the number of '1's in a picture can of course be obtained faster. Such hardware is planned for LUCAS but is not yet implemented.

An adder tree according to Figure 5.4.16 can serve this purpose. Using standard PROMs and adder circuits the summing time is 160 ns, thus smaller than the clock cycle time. The values of consecutive counts are accumulated in the "Count Accumulator", a register that can be read from the Host computer.

The time needed to count the number of '1's in a binary image is then equal to the time needed to put the bit-slices in the Tag flip-flops. This can be done at the speed of one slice per cycle. Thus, the total count time for a 128x128 image becomes 128 cycles, i.e 25.6 microseconds using a 5 MHz clock. This is ten times faster than by method 1.

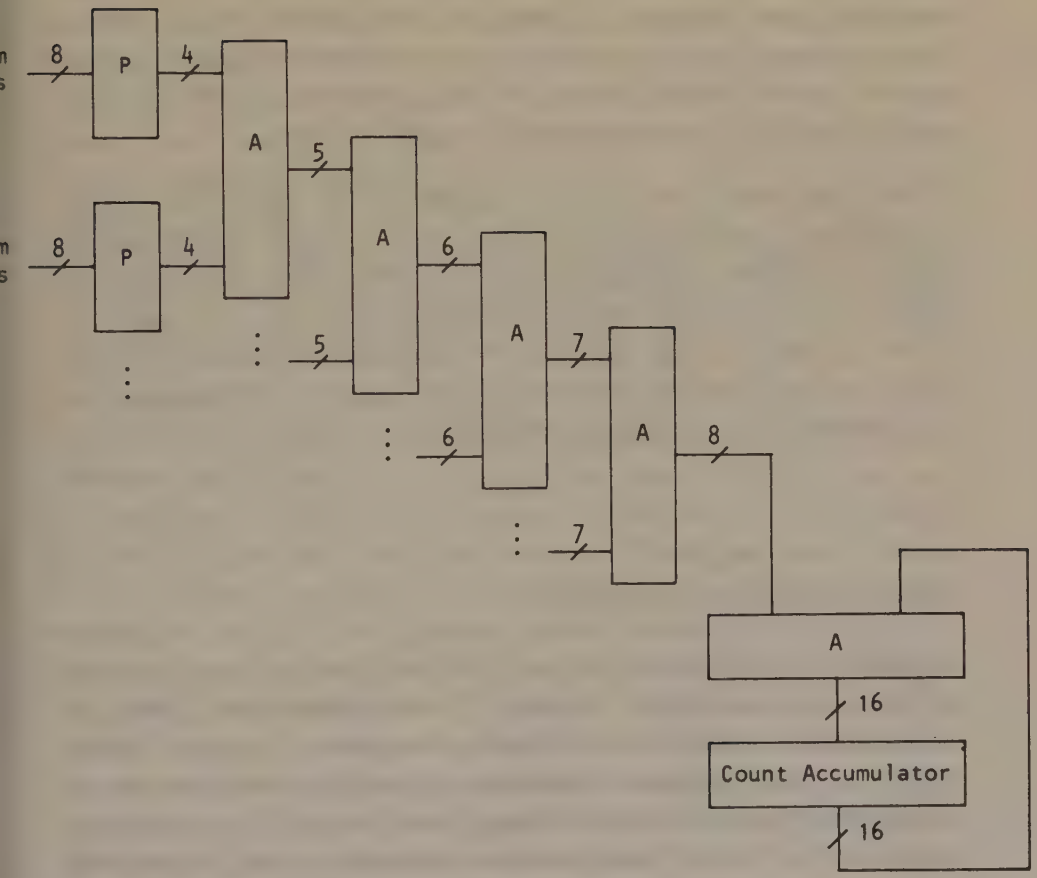


Figure 5.4.16 Part of an adder tree to count the number of responders. A=Adder, P=PROM. The total number of PROMs required is 16, the total number of 4-bit adders is 34

EXAMPLE 18: Maximum value of image

An algorithm to locate the maximum-valued element in a matrix was described in Section 2.3.4. It starts off by finding the maximum element of the first column, then examines the next column to see if there are larger elements. If there are, the largest one is taken as a new candidate, etc. A bit-slice in the associative array and a register in the Address Processor are constantly updated to keep track of where the maximum value so far can be found.

The computation time is data dependent. One search for "larger than Common" is needed for each pixel column. A search for maximum value of a column is needed for some columns. Also, data has to be moved from the array to the Common Register. In the worst case, all of this is needed for all pixel columns. Assuming 128 columns of 8-bit pixels, the worst case takes 128(12 + 29 + 16) = 7296 cycles, i.e. approximately 1.5 ms, using a 5 MHz clock.

EXAMPLE 19: Grey level histogram

Collecting the histogram of a grey level picture means counting the number of occurences of each of the possible grey levels. A straightforward method is the following: For each of the grey levels, search the entire picture and produce a binary picture with '1's in those points where the specific grey level occurs. Then count the number of '1's in the binary picture. Assuming a 128 by 128 image with 256 grey levels, the search will take approximately 1500 cycles (see section 2.3.2) and the count 1300 cycles (see example 17, method 1), i.e. 3800 cycles per grey level. Thus, the total histogram is collected in 256 x 3800 = 972 800 cycles, i.e. 195 ms using a 5 MHz clock. This is quite a long time, in fact the Host microcomputer could gather the histogram in a time that is close to this.

There are ways to shorten the time. First, using the count responders network described in example 17 will decrease the time to

133 ms. Second, the search can be made faster at the cost of having to reserve some scratch pad area for intermediate search results. The following is one possibility: Divide the grey values into four classes based on the two most significant bits. Create binary maps showing which pixels belong to each of these classes. Twelve cycles per pixel slice are needed to create these maps, i.e. 1536 cycles in total. A similar division of grey values is made based on the next two bits, etc. This gives 16 maps in total, created in 4 x 1536 = 6144 cycles. Now, the points having a certain grey value can be obtained through logical AND between four maps. This takes four cycles per bit-slice, i.e. 512 in total for each grey value. The result is obtained in the Tags, and the number of ones can be calculated in the adder tree at once. Thus the total time for histogram collection using this method will be 6144 + 256 x 512 = 137 216 cycles, i.e. 27 ms.

Histogram collection is not one of those tasks that an array of this kind performs best. With increased capabilities of the processing elements, that allows them to perform one histogram collection each on the pixels stored in their respective memories, it is possible to do well also on this task, as shown by Danielsson and Ericsson (1982). We can also choose the possibility to compute the histogram outside the array. A fairly simple device "listening" to the input or output stream of pixels can be designed for this task. Each pixel value that passes the device is used as an address pointer to a memory, and the corresponding memory word is incremented by one. The maximum I/O rate with LUCAS is one 8-bit pixel every 200 nanoseconds. A histogram collection device following this rate is realistic, and would collect a histogram for an 128 x 128 x 8 image in 3.3 ms.

5.4.6 Global transforms

There are many two-dimensional global transforms that are used in image processing, primarily for the purpose of image enhancement and restoration and image encoding. In this study we will restrict

ourselves to a brief discussion of how the two-dimensional Fourier transform can be calculated on LUCAS and the implications of this for the Walsh-Hadamard transform.

EXAMPLE 20: Two-dimensional FFT

In Section 4.3 we studied the implementation of the one-dimensional discrete Fourier transform using the FFT algorithm. The two-dimensional discrete Fourier transform of an image, I, can be obtained in the following way (Nussbaumer, 1981): First, transform each row of I to produce an intermediate matrix, G, then transform the columns of G to produce the final result, F.

With an image stored in LUCAS, transforming the rows means making an entire FFT calculation within one Processing Element. This takes considerable time, of course. However, 128 such computations are done simultaneously. Assuming a 128 x 128 image, the time for the row transforms will be the same as for the column transforms. This is because the same number of arithmetic operations are performed in the two cases, the only difference being the way data is accessed. During the row transforms, data is accessed by means of butterfly addressing within the Memory Module. In the column transforms, the shuffle/exchange network automatically provides the correct data.

The result matrix is obtained with its rows bit-reversed. When the image is output this is corrected through the use of a bit-reversed address buffer, as described in Section 4.3.

The total time for a Fourier transform of a 128 x 128 x 8 bit picture will be 256 times the required time for a 128 point one-dimensional FFT. Since this was 1.1 ms (see Section 4.3), the total time will be around 300 ms.

Another transform frequently used in image processing is the Walsh-Hadamard transform (WHT). The principle for the computation of the FFT can be applied also to the WHT (Gonzalez and Wintz, 1977).

The difference is that the trigonometric functions are reduced to plus one and minus one. This reduces the computation time with approximately 90 percent. Thus, a two-dimensional WHT could be performed on a 128 x 128 x 8 picture in 30 ms.

5.4.7 Input/output

Finally, we want to investigate how long time is needed for input/output of images.

The I/O rate of the Processor Array itself is very high. The bottleneck is outside the array. Data is transferred between the I/O data registers and the Memory Modules at a rate of 128 bits per clock cycle, i.e. 128 x 5 x 10^6 = 640 Mbits/second. However, data can not be written into or read from the I/O data registers at that speed. This is what puts the limit on I/O data speed: how fast can the rest of the system communicate with the I/O data registers?

When no special purpose I/O processor is used, the fastest way for the Host computer to communicate with the I/O data registers is through the use of the system's DMA unit. This allows filling (or reading) the 128 registers in 153.6 microseconds. The time to transfer the contents of the I/O registers to the Memory Array is 2.2 microseconds. The total time required for input/output of a 128x128 matrix of 8-bit data will then be 128(153.6 + 2.2) microseconds = 19.9 ms. A binary image requires 1/8 of this time, i.e. 2.5 ms.

If we imagine an I/O processor capable of writing or reading an I/O register with 5 MHz, we could fill the 128 I/O registers in 25.6 microseconds. The time required for a 128 x 128 x 8 image would then be 128(25.6 + 2.2) microseconds = 3.6 ms. A binary image would require 0.45 ms.

One further comment on the input/output time should be made: Filling (or reading) the I/O data registers from the Host computer or

I/O processor can be done at the same time as computations take place in the array (provided that the computations do not use communication over the I/O data registers, which they have not done in any of the examples examined). Thus, for tasks that are computation bound, the effective input/output time is in fact 2.2 microseconds per 8-bit slice, i.e. 282 microseconds for a whole 128 x 128 x 8 bit image.

5.4.8 Larger images

Throughout Section 5.4 we have assumed that the size of the image side agrees with the number of Processing Elements, so that one line of the image exactly occupies a field of the memory.

When the number of pixels per line in the image is greater than the number of PEs we propose that each PE takes care of more than one column of the image. For example, a 512 x 512 pixels image is stored with four columns per Memory Module. We propose neighbouring columns because this is advantageous from neighbourhood access point of view. (Larger accessible neighbourhood). Each memory module would receive 512 x 4 = 2048 pixels. Since the MMs are only 4096 bits wide, only two bits per pixel can be stored. This means that LUCAS is not large enough to hold larger pictures than that. There is the possibility of taking in a part of the image at a time. For some operations this is certainly feasible. In most cases, however, it is impractical. To equip this kind of machine with larger memories is one of the easiest things to do and we feel it is highly recommendable if the machine is to be used for image processing.

We disregard the memory length problem for a while and concentrate on how the pixels should be individually ordered within the MM. We take as example a 16 x 16 pixels image to be stored in a 4 PE machine. Figure 5.4.17 shows which pixels of the image are stored in each Memory Module. We propose a storage ordering according to Figure 5.4.18. It makes the addressing to access neighbouring pixels simple. For each pixel we have that its eight

nearest neighbours are stored at the pixel places with addresses -16±1, 0±1 and +16±1 relative to the pixel's own address and taken modulus64. Some neighbours are in the same MM, others in a neighbouring one.

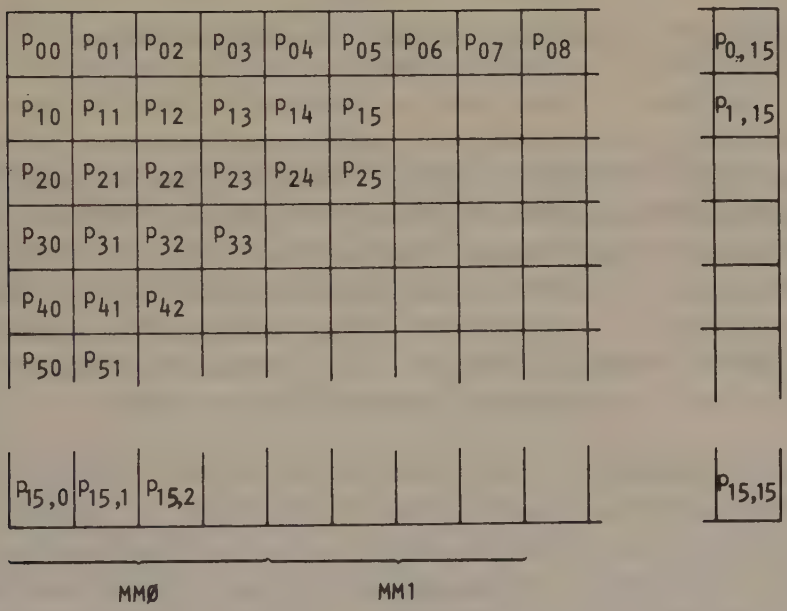


Figure 5.4.17 Division of a 16x16 image on four Memory Modules

MM0:	$P_{00}\ P_{10}\ P_{20}\ P_{30}\cdots\ P_{01}\ P_{11}\ P_{21}\ P_{31}\cdots\ P_{02}\ P_{12}\ P_{22}\ P_{32}\cdots\ P_{03}\ P_{13}\ P_{23}\ P_{33}\cdots$
MM1:	$P_{04}\ P_{14}\ P_{24}\ P_{34}\cdots\ P_{05}\ P_{15}\ P_{25}\ P_{35}\cdots\ P_{06}\ P_{16}\ P_{26}\ P_{36}\cdots\ P_{07}\ P_{17}\ P_{27}\ P_{37}\cdots$
MM2:	$P_{08}\ P_{18}\ P_{28}\ P_{38}\cdots\ P_{09}\ P_{19}\ P_{29}\ P_{39}\cdots$
MM3:	$P_{0,12}\ldots$

Figure 5.4.18 Storage order of pixels within Memory Modules

Input/output according to these principles is not without problems. When the number of pixels per line agrees with the number of PEs, pixels arriving one by one in TV scan mode are just written into the I/O registers in the order of arrival. Now, every fourth pixel only - no. 0, 4, 8 and 12 - are to be put in the registers. When these have been input to the array, pixels no. 1, 5, 9 and 13 are treated in the same way, etc. The procedure is repeated for each line.

What is needed is a device with enough storage to store a line and with addressing hardware that can read out the contents in another order than the one in which it was stored. In the case that served as an example, the address bits are merely shifted two steps to the left, giving the sequence 0,4,8,... when the two rightmost bits are 00, the sequence 1,5,9,... when they are 01, and so on. Thus, this can be a very simple device.

If the device shall be able to handle different ratios between image size and array size it should probably be designed as a microprogrammable address generator. Then, different microprograms, giving different address sequences, could be initiated depending on the ratio at hand.

5.4.9 Comparison of execution times

VAX 11/780

In order to get an idea of what the reported processing times are worth we have programmed a few of the algorithms on a conventional VAX 11/780 computer. The programs were written in Vax assembly language. The comparison is summarized in Table 5.4.1.

As can be expected, the greatest difference in time is found in the case of binary images. The simple shrinking operation takes 650 times longer time on the VAX computer than on LUCAS. With 8-bit data the VAX computer is better off, but LUCAS is still nearly two orders of magnitude faster. 16-bit pixel values are not very common in image processing. Comparison between 8- and 16-bit processing times show that an ordinary computer like VAX cannot take advantage of the fact that image data have low precision - the processing times for 8- and 16-bit data are nearly identical.

Binary image	Time on VAX (ms)	Time on LUCAS (ms)	Ratio
Border8, Shrink8	130	0.2	650
8-bit pixel values			
Laplace L_4	197	2.61	75
Laplace L_8	290	5.23	55
Roberts' cross-difference	218	3.64	60
Mean value 3x3	335	4.28	78
16-bit pixel values			
Laplace L_4	203	4.46	46
Laplace L_8	296	8.92	33
Roberts' cross-difference	218	6.80	32
Mean value 3x3	374	7.35	51

Table 5.4.1 Compared processing times for VAX 11/780 and LUCAS. Image size is 128 x 128 pixels

For some of the special purpose image processing machines that we have mentioned in this chapter, performance results from implemented image operations have been reported. We will take a few such examples. Since we want to use the results to make comparisons with the processing times on LUCAS, we have only chosen such results that can be put directly in relation to LUCAS results.

DAP

Marks (1980) gives the time needed for collection of histogram on the pilot DAP with 32 x 32 PEs and 200 ns cycle time. The histogram of a 192 x 192 pixels image with 6-bit grey scale is obtained in 17.25 ms.

To get a comparative measure for LUCAS, we imagine a 128 x 256 pixels image with 6-bit pixel values. This is very close in size to the one Marks uses. LUCAS would require 10.8 ms to collect the histogram, provided it was equipped with an adder tree. Without an adder tree the time would be 44 ms. We note that the DAP has eight times as many processors as LUCAS.

Marks further reports processing time for the following operation on an image of the same size: The image is first differentiated in two directions, the absolute values are formed, thresholding performed, and logical OR between the results is taken. The time for this is reported to be 2.9 ms.

On LUCAS, the same operation on a 128 x 256 x 6 bit image would take 5.8 ms, i.e. twice as long time. LUCAS has more powerful instructions in the PEs which probably in part accounts for the ratio being smaller than eight, which is the ratio between the numbers of processors in the two machines. Also, the addressing of neighbouring pixels within the PEs causes some overhead in DAP.

CLIP4

Fountain and Goetcherian (1980) report execution times for a couple of algorithms implemented on CLIP4. Addition of two images, 96 x 96 x 16 each, takes 450 microseconds on CLIP4. LUCAS adds two 128 x 128 x 16 images in 1332 microseconds. A way of comparing the measures is to look at the time per pixel, which is 49 ns for CLIP4 and 81 ns for LUCAS. Thus CLIP4, with its 72 times as many processing elements, is only 40 % faster per pixel.

An edge detection algorithm for binary 96 x 96 pictures, similar in complexity to a shrinking operation, is reported to take 25 microseconds on CLIP4. Binary shrinking of a 128 x 128 image on LUCAS takes 180 microseconds. This is 2.7 ns/pixel for CLIP4 and 11 ns/pixel for LUCAS. Thus, in this case CLIP4 can be considered 4 times as fast.

Picap-FIP

The main features of the Picap-FIP processor are the use of four special purpose processors operating in parallel and the utilization of a fast cache memory to hold that portion of the image that is currently treated. In (Kruse et al, 1980) the execution time for Roberts' cross difference operator performed on Picap-FIP is given. The time required is 100 ns/pixel (8-bit data). On a 128 x 128 pixels image, this makes 1.6 ms. The comparative time for LUCAS is 3.64 ms.

FLIP-FIP

The FLIP-FIP, using 16 identical processors, is reported to perform median filtering over a 3x3 neighbourhood in 1 second for a 512 x 512 pixels image (Gemmar et al., 1981). This makes 3.8 microseconds/pixel. On LUCAS, the same operation is performed on a 128 x 128 image in 70 ms, which makes 4.2 microseconds/pixel.

Laplace-filtering using 3x3-window is reported to take 0.2 seconds for a 512 x 512 image on FLIP-FIP. This makes 0.76 microseconds/pixel. On LUCAS, a 128 x 128 image is treated in 2.61 ms, which makes only 0.15 microseconds/pixel.

Conclusion

We note that the processing times presented for LUCAS and those for the other machines are of the same order of magnitude. The comparisons with VAX show that the times are about two orders of

magnitude shorter than the times on a sequential computer. We take these figures as an indication that LUCAS has the potential to be a useful tool in image processing.

5.5 CONCLUSIONS

As we noted at the beginning of this chapter, image processing is a large computational area with many different demands. The processing examples that we have treated in this chapter by necessity cover but a small part of the types of computations that an image processing system should be able to perform efficiently. The presented operations are all examples of tasks that require very long execution times when performed on conventional computers. We have shown that they can be solved on LUCAS with a considerable speed-up compared to sequential execution.

More important than the usefulness of the physical machine is the usefulness of the kind of architecture that it represents. We feel quite convinced that there is a need for bit-serial processor arrays in image processing. LUCAS represents another kind of array than DAP, CLIP4 and MPP, with a number of PEs that is in the order of the squareroot of the image size instead of in the order of the image size itself. Our experience is that using a number of PEs that is equal to the image side and organizing the PEs in one dimension only, give very straightforward programming and simple input/output.

If varying image sizes are used, this organization may have some drawbacks, and it may be favourable to use a two-dimensional organization, like in a recent proposal from Linkoping, called LIPP (Danielsson and Ericsson, 1982). The two-dimensional organization gives a more intricate neighbourhood addressing scheme and thus puts stronger demands on the address generating control unit.

CHAPTER 6
CONCLUSIONS AND FUTURE RESEARCH

In this chapter we will give a summary of the results obtained. We will also report some experiences drawn from the project. A project of this kind, including everything from the first architectural ideas to the actual use of the machine in applications, offers many experiences of the most varying kind. The results and experiences will be grouped under four headings: design, implementation, applications, and programming. In the design part we will review the design considerations made and suggest modifications that would improve the performance. In the section on implementation we mention some experiences from building the machine, but primarily we treat the possibilities of using VLSI technology to implement this kind of computer structure. The application part is a summary of the application results reported in Chapters 4 and 5 but also includes some general remarks and suggestions for other areas. Finally, the experiences of programming LUCAS are discussed briefly.

6.1 DESIGN

Processing Elements

The bit-serial, word parallel working mode is the prime characteristic of the processor array. We have found that great flexibility and generality is offered by the use of bit-serial processing elements. Treating many bits in parallel in each PE would of course give faster processing in many cases, but often that kind

of parallelity could not be utilized. It would also be more difficult to decide on an instruction set for the PEs in the bit-parallel case.

The Processing Elements have been found to have the necessary facilities for most tasks, with respect to both the number of flip-flops and the available functions. Sometimes - but surprisingly seldom - the processing would have been faster if more boolean functions had been available.

A minor change that would have improved the performance on some tasks is the following (see Figure 1.3.1): If the Direct input (D) and the Common input (COM) were interchanged it would still be possible to input one bit from each source simultaneously. But it would also be possible to input one bit on D and at the same time one bit on, say, the "Above" input which would make vertical differentiation faster.

To increase the processing speed of a processor array there are two ways to follow. One is to increase the number of processors. The other is to make the processors more powerful, which can be done without abandoning the bit-serial working mode.

There are application areas where the first approach is advantageous. Data base processing is probably such an area. However, image and signal processing may benefit more from improving the power of the processors. We will discuss a few such improvements.

Multiplication is a function often needed. It can be speeded up if a shift register is used in the processing elements to hold the partial products. This is included in both MPP (Batcher, 1982) and PROPAL 2 (Cimsa, 1979). Furthermore, as we noted in some examples on image processing, a counter that could be incremented or decremented in one clock cycle would add significantly to the performance. The counter function could be integrated with an index register function. The latter would be useful to "shift" data a different number of bits in different memory words - necessary e.g. in floating point operations - and also for table look-up.

6.1.2 Communication

Perhaps the most difficult decision to make when designing a processor array is which interconnection network to use. One of the purposes of the project was to test different structures. Therefore the network was made rewirable. When analysing the possibilities for parallel solution of a certain problem, we believe it is a great advantage not to be constrained to map the problem onto a certain interconnection structure.

It is quite probable that if machines of this type were produced, they would most often be used for specific applications. Therefore the possibility to select different interconnections in different installations could be valuable. In LUCAS this is achieved by outputting one bit from each memory word to a common connector from which all PEs can choose their inputs. The use of particular "interconnection boards" to define the topology of the array constitutes an attractive possibility for future implementations.

In the applications described in this dissertation, nearest neighbour communication in one dimension together with a perfect shuffle/exchange network has been used. With this network the longest path between any two processors is $\log_2 N$, where N is the number of PEs. With two-dimensional interconnection, as in DAP and MPP, the longest distance is $\sqrt{N}$, which grows much faster than $\log_2 N$.

In the application programs written, the delay caused by routing of data between processors is generally very small, often negligible. Our experience of using the perfect shuffle/exchange network is positive. However, the experience is yet too limited in order for us to say anything definite about its superiority to other proposals. The bit-serial working mode of the PEs is an important pre-requisite for keeping the network within manageable size.

Input and output of data constitutes another communication problem. From one aspect this problem is the same for processor arrays as for conventional computers: how do we get data into or out of the memory as fast as possible. However, there are differences: In

the first place, the time for input and output becomes more noticeable when the processing is faster; we may find it unsatisfactory to have longer input/output time than processing time. In the second place, there may be certain demands on how the data is stored in the memory. The storage pattern must suit the processing that is to take place under the constraints of the interconnection structure.

As for speed, it is important to be able to perform input/output and processing simultaneously. In LUCAS this is done by letting the low-bandwidth part of the data transfer - the passing of words between the I/O data registers and the Host computer - overlap the processing of data. The speed limit of this transfer is set by the Host. Therefore it can be stated that the input/output speed that LUCAS offers is high enough for communication with other computers and with conventional input/output devices. In order to increase the speed further, e.g. in order to do real time image processing, a special purpose I/O processor must be used. This would account for filling/reading the I/O data registers faster. The I/O processor could also take care of the the other task: that of reordering data to suit the computations in the array. An I/O processor of this kind is now under development for LUCAS.

The I/O data registers of LUCAS serve two purposes. Their primary use is input/output but they can also be used for transfer of data between different memory words, or between the memory and the Common Register. In applications where the latter kind of use is frequent we suggest the implementation of two registers per word, since otherwise input/output and processing could not be overlapped. Doubling the I/O data registers can also be motivated by desires to overlap the input of data with the output of data, which might be required in some applications.

6.1.3 Control Unit

It is very important that the Control Unit be powerful and flexible. This was realized at an early stage of the project. The importance of a powerful processor dedicated to the computation of bit-slice addresses for the memory array cannot be overestimated.

The Address Processor implemented in the Control Unit of LUCAS has proved to be just powerful enough to provide bit-slice addresses at the necessary speed. We find it an attractive alternative if the Address Processor in addition to this could take care of some of the tasks that are now performed by the Host computer. This could include e.g. the management of tables of pointers to data in the memory array.

In one of the application examples in Chapter 5 we noted that it would be beneficial if we were not constrained to using the same bit address for the Common Register as for the Memory Modules. Probably, an index register for the address to the Common Register would be sufficient.

6.2 IMPLEMENTATION

6.2.1 Experience

LUCAS has been built with limited resources in a university environment. Initially we lacked much knowledge of "practical" kind that is usually considered necessary for the implementation of large systems. The fact that we have got the system working proves that the difficulties of getting hundreds of processing elements operate in parallel are not insurmountably large.

Of course there have been problems. The step from a reliable system with 32 PEs to a reliable one with 128 PEs was much more difficult to take than we expected. One of the main obstacles was the noise produced when all memory modules changed output simultaneously

- but not exactly simultaneously due to differences in circuit speed and and address delay between different words. The problem had its simple solution - disabling the outputs until they were all stable, which could be made through a change in the microprograms.

An important feature of today's technology as compared to the technology at hand when the von Neumann computer model was suggested is that memory and processing logic are now made using the same technique. Therefore there is no reason to distinctly separate memory from logic. In other words, from a pure technological point of view the use of data processing memory is reasonable. In a processor of LUCAS kind the distinction between memory and processing logic is not as distinct as in sequential computers. This suggests that the use of large scale integration technology has extraordinary advantages in such processors. We will next consider the possibilities available.

6.2.2 Possibilities offered by VLSI technology

No doubt, the kind of processor that we discuss in this thesis is very well suited for VLSI implementation. As part of a multi-project chip the logic of one Processing Element (excluding memory) was in fact implemented in CMOS/SOS by the project group in 1981.

Let us investigate what the consequences of integrating many processing elements on one chip would be in terms of number of gate functions and number of pins per chip.

Off-chip memory

We first consider the consequences of using ordinary read/write memory chips for the memory modules. We further assume that we want exactly the facilities now implemented in LUCAS. This means that for interconnection each processing element needs one input from above, one from below, one for shuffle and one for shuffle+exchange. We

further assume that the control signals for IO data registers, multiplexer and ALU functions are gathered to a single "instruction code", k bits wide. 8 bits would be appropriate to implement the present possibilities. Finally, we assume b bits wide I/O data registers. In LUCAS, b is 8.

Table 6.2.1 describes the number of pins needed on a chip comprising n processing elements. Table 6.2.2 lists the function for different values of n for two different combinations of b and k.

The first combination, b=8 and k=8, represents what is implemented on the current LUCAS. b=16 means improving the I/O rate by a factor of two. k=12 means increasing the number of ALU functions significantly.

Specification	pins
I/O data bus	b
I/O write	1
I/O data register address	$\log_2 n$
Chip select	1
Select First chain	2
Data in/out	n
Common Register output	1
Shuffle input	n
Above/Below	2
Instruction code	k
Power,Ground,Clock	3
	Sum: $b+k+10+2n+\log_2 n$

Table 6.2.1 The number of pins of an n processor chip

	(I)	(II)
	b=8 k=8	b=16 k=12
n=1	28	40
n=2	31	43
n=4	36	48
n=8	45	57
n=16	62	74
n=32	95	107
n=64	160	172

Table 6.2.2 The number of pins for different values of n assuming (I) 8-bit data and 8-bit instruction code and (II) 16-bit data and 12-bit instruction code

The number of gate functions needed to implement the processors is totally dominated by the logic required to implement the arithmetic/logic unit. Assuming k_1 bits, out of the k instruction bits, are needed to specify the function and assuming f flip flops in each PE, one bit from Memory and one from Common,

$$G_1 = f * 2^{f+k_1+1+1}$$

memory cells are needed to implement the ALU of one PE as a ROM. (This is an upper limit, since the number can be reduced considerably if a PLA structure is used instead of a ROM.) In LUCAS, we have f=4 and k_1=5, which gives

$$G_1(\text{LUCAS}) = 4 * 2^{11} = 2^{13}$$

In an n processor chip, nG_1 memory cells are required. Table 6.2.3 lists nG_1 for different values of f and k_1.

Using these tables we can now choose parameters and a value of n that give values of cell count and number of pins within the limit of available technology.

nG_1	f=2	f=4	f=4
	k_1=4	k_1=5	k_1=7
n=1	2^9	2^{13}	2^{15}
n=2	2^{10}	2^{14}	2^{16}
n=4	2^{11}	2^{15}	2^{17}
n=8	2^{12}	2^{16}	2^{18}
n=16	2^{13}	2^{17}	2^{19}
n=32	2^{14}	2^{18}	2^{20}
n=64	2^{15}	2^{19}	2^{21}

Table 6.2.3 The number of memory cells required for an n-PE chip with f flip-flops per PE and 2^{k_1} functions

For example

$b=8,\ k=8,\ f=4,\ k_1=5$

as in LUCAS, would make it possible to put 32 PEs in a 95 (or maybe 96) pin chip comprising 2^{18} (=256k) cells which is possible with current VLSI technology.

We assumed the memory modules were outside these chips. Suppose we want a memory word length of 64 kbits. We could then use memory chips of 64 K 8-bit words. Four of these would be required to support one 32-PE chip of the above kind. A circuit board with 80 chips would then have 512 Processing Elements, each with 64 kbits of memory. Some additional chips for I/O address decoding and buffering would be needed on each board.

A particular problem appears with the perfect shuffle/exchange network - if this is the one chosen. We would like to be able to use many of the 512-PE boards together, and be able to perform shuffle permutation on the total of PEs. Can this be implemented without rewiring the whole network when new boards are added?

1024 PE shuffle permutation can be performed in twice the time if the two boards can exchange data over a 512 bit bus. In general, if m boards are used, a 512xm shuffle can be made in a time of m shuffles. (It is assumed that an individual PE can choose the shuffle or the shuffle/exchange input based on e.g. the tag contents).

On-chip memory

We next consider the case of including the memory modules in the PE chips. If we want to equip the PEs with index registers for addresses, this alternative should probably be chosen. Otherwise, the PEs must output the memory addresses.

To provide an address for the bit slice of the memory, address pins are needed. We assume 2^m-bits memory words, requiring m address bits. Furthermore, a write control signal is needed. Thus, the pin count exceeds what we had in table 6.2.2 with m+1.

With m=16, the 16 processors/chip case would require 79 pins/chip. A board with 64 such chips would thus have 1024 processors in total.

The number of gates needed for the memory modules would dominate the gate count in such a chip. Assuming n processors with 2^m bits of memory each, $n2^m$ memory cells are needed. For n=16 and m=16, this makes 2^{20}, which is one million. The cell count for the PE part, according to table 6.2.3, ranges between 2^{12} and 2^{18}, depending on complexity.

We conclude that, with memory on the chip, it is probable that it is the number of gate functions that puts the limit on how many processors can be implemented on one chip.

Before leaving this example we also point to the attractive possibility of using read/write memory to implement the arithmetic/logic unit. Loading of the ALU memory can e.g. be done using the memory address pins and I/O data pins.

No interconnection network

In the cases considered above we have assumed that communication is needed between processing elements. In some applications this is not required. Relational data base management (Kruzela, 1983) is the prime example. We end up by considering the consequences of this for VLSI implementation.

The number of pins required for an n-PE chip will be (cf. Table 6.2.1)

$$b+k+10+\log_2 n+m+1$$

where b is the I/O data bus width, k is the instruction code length, and m is the memory address length. (We assume memory on chip).

Table 6.2.4 lists this number for different values of the parameters. We can see that the pin count is very low, even with very many PEs on the chip.

	b=8 k=8 m=12	b=16 k=8 m=16	b=32 k=8 m=16
n=1	39	51	67
n=4	41	53	69
n=16	43	55	71
n=64	45	57	73
n=256	47	59	75
n=1024	49	61	77

Table 6.2.4 The number of pins of an n-processor chip without external interconnection. b is the I/O data width, k is the length of the instruction code, and m is the length of the memory address.

Clearly, it is the number of gate functions that puts the limit on what is implementable in this case. The number of memory cells per processor is 2^m for the memory words and $f * 2^{f+k_1+1}$ for the PEs. Assuming f=4, i.e. each PE having four flip-flops, the two counts have the same value if $m=k_1+7$. Assuming f=2, which is probably sufficient for data base processing, the two counts are equal if $m=k_1+4$. $k_1=4$ is probably sufficient in this case. Thus, we conclude that the number of gate functions to implement the memory modules will again be totally dominating. Assuming we can have 2^{20} (1 million) memory cells on a chip, and that the word length is chosen to be 2^{12} bits. Then 256 processors could be implemented on a single chip. A data base processor with thousands of processing elements would be easily implemented with these circuits.

6.3 APPLICATIONS

In the course of the project we have found that the range of applicability is wider than we first expected. We have found effective solutions to many problems that were not considered when the architecture was decided. Surprisingly, it has often been very easy to map the problems on the structure of the machine, contradicting the opinion that a parallel architecture with a parallel interconnection scheme is effective only on a very limited range of applications.

In Chapter 4 we studied the use of LUCAS in three different classes of computations, namely matrix multiplication, discreté Fourier transformation and the solution of graph theoretic problems. Each one is important enough to motivate the design of special purpose hardware for its execution, if necessary. We found that we could map the data structures involved onto LUCAS in a natural way and get efficient solutions of the problems, using the full parallelism of the hardware.

An interesting fact about the three computational fields studied in Chapter 4 is that the demands put on the hardware differ very much from one problem to the other. We showed that it is possible to satisfy the diverse demands well with one single architecture.

In Chapter 5 we discussed different methods of meeting the specific requirements of image processing. The area is large and diverse and we should not expect that one single architecture is suited for all computations. We have given arguments that SIMD processors of bit-serial type offer great advantages especially for performing local operations on images, and we have shown this by programming some typical algorithms on LUCAS and measuring the performance. An important feature is the possibility to treat pixel values of different length with full efficiency. Short operands are very common in image processing. The mapping of images to the memory array that we are using - one column of the image per memory word - is simple from input/output point of view and has also proved practical when microprogramming the algorithms.

We have looked not only on local image operations but also on global transforms, which are computed efficiently, and on image measurements. The performance of the latter can be improved significantly if some rather simple additional hardware is used.

The applications shown in this thesis represent a wide range of problems. In addition to this, the use of LUCAS-type processors for data base processing has been studied with promising results (Kruzela, 1983). Most important, it has been shown that considerably more involved operations can be performed efficiently than the mere searches traditionally considered when the use of associative processors has been discussed. Yet other areas will be studied on LUCAS. Promising applications include sequential decoding problems, imaging using Nuclear Magnetic Resonance technology and real time image processing in automatic control systems.

6.4 PROGRAMMING

One of the conclusions that we have drawn from the programming of LUCAS is: it is easy. Most of the programming that we have done has been on the microprogramming level. We are quite convinced that microprogramming of ordinary, sequential computers presents more intricate problems than those encountered when microprogramming LUCAS. Most difficult has in fact been to control the Address Processor, i.e. to step the field pointers and loop counters appropriately. After the high level microprogramming language had been developed, this "sequential" part of the programming was automatized which made microprogram development much faster.

Many problems are inherently parallel and the solutions to them are more naturally found and described using parallel languages. In existing sequential languages, processing of arrays is expressed using various indexing and loop structures. This often makes programs difficult to understand (and write). The corresponding expressions in a parallel language are closer to the original thought and are easier to understand.

Our experience from parallel languages is of course very restricted since we have not had the possibility of actually using one. However, the Pascal/L language that has been specified (Fernstrom, 1983) has proved useful for expressing and communicating algorithms actually implemented as microprograms. We consider further development of language constructs and programming tools to be one of the most important future research topics.

REFERENCES

Backus, J. (1978) "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM, vol. 21, no. 8, pp. 613-641.

Baer, J.L. (1980) "Computer Systems Architecture", Pitman, London.

Barnes, G.H., Brown R.M., Kato K., Kuck D.J., Slotnick D.L., and Stokes R.A. (1968) "The ILLIAC IV computer", IEEE Transactions on Computers, vol. C-17, no. 8, pp. 746-757.

Batcher, K.E. (1974) "STARAN parallel processor system hardware", Proc of the 1974 National Computer Conference, pp. 405-410.

Batcher, K.E. (1976) "The flip network in STARAN", Proc of the 1976 International Conference on Parallel Processing, p. 65-71.

Batcher, K.E. (1977) "The multidimensional access memory in STARAN", IEEE Transactions on Computers, vol. C-26, no. 2, pp. 174-177.

Batcher, K.E. (1979) "The STARAN computer", in "Supercomputers", Infotech State of the Art Report.

Batcher, K,E. (1980) "Design of a massively parallel processor", IEEE Transactions on Computers, vol. C-29, pp. 836-840.

Batcher, K.E. (1982) "Bit-serial parallel processing systems", IEEE Transactions on Computers, vol. C-31, pp. 377-384.

Bentley, J.L. (1979) "A parallel algorithm for constructing minimum spanning trees", Seventeenth Annual Allerton Conference on Communication, Control, and Computing, pp. 11-20.

Cimsa (1979) "Processeur Parallelle Associatif - PROPAL 2. Presentation", Cimsa - Compagnie d'Informatique Militaire Spatiale et Aeronautique, 781 40 Velizy, France (in French).

Cooley, J.W., and Tukey J.W. (1965) "An algorithm for the machine calculation of complex Fourier series", Math. of Compu., vol. 19, pp. 297-301.

Danielsson, P.E., and Ericsson T. (1982) "Suggestions for an image processor array", Internal Report LiTH-ISY-I-0, Linköping University, Sweden.

Danielsson, P.E. (1981) "Getting the median faster", Computer Graphics and Image Processing 17, pp. 71-78.

Danielsson, P.E., and S. Levialdi (1981) "Computer architectures for pictorial information systems", Computer, November, pp. 53-67.

Dijkstra, E.W. (1959) "A note on two problems in connection with graphs", Numerische Math., vol. 1, pp. 269-271.

Duff, M.J.B. (1979) "Parallel processors for digital image processing", in "Advances in Digital Image Processing", edited by P. Stucki, Plenum Press, New York, pp. 265-276.

Duff, M.J.B., and Levialdi S., (editors) (1981) "Languages and Architectures for Image Processing", Academic Press, London.

Fernström, C. (1982) "Programming techniques on the LUCAS associative array computer", 1982 International Conference on Parallel Processing, Bellaire, Michigan, Aug.

Fernström, C. (1983) "The LUCAS Associative Array Processor and its Programming Environment", Ph.D. dissertation, Department of Computer Engineering, University of Lund.

Flanders, P.M., Hunt D.J., Reddaway S.F., and Parkinson D. (1977) "Efficient high speed computing with the Distributed Array Processor", in High Speed Computer and Algorithm Organization, edited by D.J. Kuck, D. Lawrie, and A.H. Sameh, Academic Press, New York.

Floyd, R.W. (1962) "Algorithm 97: shortest path", Comm ACM, vol. 5, p. 345.

Flynn, M.J. (1966) "Very high-speed computing systems", Proc of IEEE, vol. 54, no. 12, pp. 1901-1909.

Fountain, T.J., and Goetcherian V. (1980) "Clip parallel processing system", IEEE Proceedings, vol. 127, Pt.E., No. 5, pp. 219-224.

Gemmar, P., Ischen H., and Luetjen K. (1981) "FLIP: A multiprocessor system for image processing", in (Duff and Levialdi, 1981), pp. 245-256.

Gonzalez, R.C., and Wintz P. (1977) "Digital Image Processing", Addison-Wesley, Reading, Massachusetts.

Goodyear Aerospace Corporation (1976) "Digital image processing and STARAN", Report GER-16336, GAC, Akron, Ohio.

Granlund, G. (1981) "GOP: A fast and flexible processor for image analysis", in (Duff and Levialdi, 1981), pp. 179-188.

Hockney, R.W., and Jesshope C.R. (1981) "Parallel Computers: Architecture, Programming and Algorithms", Adam Hilger Ltd, Bristol.

Hwang, K. (1979) "Computer Arithmetic: Principles, Architecture and Design", Wiley, New York.

IEEE G-AE Subcommittee on Measurement Concepts (1967) "What is the fast Fourier transform?", IEEE Transactions on Audio Electroacoustics, AU-15(2), pp. 45-55.

Iliffe, J.K. (1982) "Advanced Computer Design", Prentice Hall, London.

Justusson, B.I. (1980) "On the use of medians and other order statistics in picture processing", Proceedings of the First Scandinavian Conference on Image Analysis, Linköping, pp. 84-86.

Kruse, B. (1973) "A parallel picture processing machine", IEEE Transactions on Computers, C-22, pp. 1075-1087.

Kruse, B. (1977) "Design and Implementation of a Picture Processor", Ph.D. dissertation, Linköping Studies in Science and Technology Dissertation, no. 13.

Kruse, B., Gudmunsson B., and Antonsson D. (1980) "FIP - the picap II filter processor", 5th International Joint Conference on Pattern Recognition, pp. 484-488.

Kruzela,I. (1980) "LUCAS Microprogramming Manual", Department of Computer Engineering, University of Lund, December 1980.

Kruzela, I. (1983) "An Associative Array Processor Supporting a Relational Algebra", Ph.D. dissertation, Department of Computer Engineering, University of Lund.

Kruzela, I., and Svensson B. (1981) "The LUCAS architecture and its use in relational data base processing", Sixth Workshop on Computer Architecture for Non-Numeric Processing, Hyeres, France.

Kushner, T., Wu A., and Rosenfeld A. (1980) "Image processing on ZMOB" Technical Report TR-987, Computer Vision Laboratory, University of Maryland, College Park, MD, USA.

Kushner, T., Wu A., and Rosenfeld A. (1981) "Image processing on MPP:1", Technical Report TR-1007, Computer Vision Laboratory, University of Maryland, College Park, MD, USA.

Lawrie, D.H. (1975) "Access and alignment of data in an array processor", IEEE Transactions on Computers, vol. C-24, no. 12, pp. 1145-1155.

Lenfant, J. (1978) "Parallel permutation of data: A Benes network control algorithm for frequently used permutations", IEEE Transactions on Computers, vol. C-27, pp. 637-647.

Loucks, W.M., Snelgrove M., and Zaky S.G. (1982) "A Vector Processor Based on One-Bit Microprocessors", Computer, February, pp. 53-62.

Marks, P. (1980) "Low-level vision using an array processor", Computer Graphics and Image Processing, vol. 14, pp. 281-292.

Mick, J., and Brick J. (1980) "Bit-slice Micro-processor Design", McGraw-Hill.

Nussbaumer, H.J. (1981) "Fast Fourier Transform and Convolutional Algorithms", Springer-Verlag, Berlin.

Ohlsson, L. (1982) "Real time spectral analysis of speech on a small associative computer", Master Thesis Technical Report, Department of Computer Engineering, University of Lund.

Ohlsson, L., and Svensson, B. (1982) "Matrix multiplication on LUCAS", Technical Report, Department of Computer Engineering, University of Lund.

Parker, D.S. (1980) "Notes on shuffle/exchange-type switching networks", IEEE Transactions on Computers, vol. C-29, no. 3, pp. 213-222.

Pease, M.C. (1968) "An adaptation of the fast Fourier transform for parallel processing", Journal of the Association for Computing Machinery, vol. 15, no. 2, pp. 252-264.

Pease, M.C. (1977) "The indirect binary n-cube microprocessor array", IEEE Transactions on Computers, vol. C-2 , no. 5, pp. 458-473.

Potter, J.L. (1978) "The STARAN architecture and its application to image processing and pattern recognition algorithms", National Computer Conference, pp. 1041-1047.

Rao, C.V.K., Prasada B., and Sarma K.R. (1976) "A parallel shrinking algorithm for binary patterns", Computer Graphics and Image Processing, vol. 5, pp. 265-270.

Reddaway, S. (1979) "The DAP approach", Infotech State of the Art Report on Super Computers, vol. 2.

Rieger, C., Bane J., and Trigg R. (1980) "ZMOB: A highly parallel multiprocessor", 1980 IEEE Workshop on Picture Data Description and Management, pp 298-304.

Roberts, L.G. (1965) "Machine perception of three-dimensional solids", in "Optical and Electrooptical Information Processing" (J.T. Tippett et al., eds.), pp.1 59-197, MIT Press, Cambridge, Massachusetts.

Rosenfeld, A., and Kak A.C. (1976) "Digital Picture Processing", Academic Press, New York.

Siegel, H.J. (1981) "PASM: A reconfigurable multimicrocomputer system for image processing", in (Duff and Levialdi, 1981), pp. 257-265.

Siegel, H.J., Siegel L.J., McMillen R.J., Muller, Jr. P.T., and Smith S.D. (1979) "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition", 1979 Computer Society Conference on Pattern Recognition and Image Processing, pp. 214-224.

Siegel, H.J., and Smith S.D. (1978) "Study of multistage SIMD interconnection networks", Proceedings of the 5th International Symposium on Computer Architecture, pp. 223-229.

Stone, H.S. (1971) "Parallel processing with the perfect shuffle", IEEE Transactions on Computers, vol. C-20, no. 2, pp. 153-161.

Swain, P.H., Siegel H.J., and El-Achkar J. (1980) "Multiprocessor implementation of image pattern recognition: A general approach", 5th International Joint Conference on Pattern Recognition, pp. 309-317.

Unger, S.H. (1958) "A computer oriented towards spatial problems", Proceedings of IRE, vol. 46, pp. 1744-1750.

Yew, P-C., and Lawrie D.H. (1981) "An easily controlled network for frequently used permutations", IEEE Transactions on Computers, vol. C-30, no. 4, pp. 296-298.

APPENDIX

A1. SCHEMATIC DRAWINGS OF A PROCESSOR BOARD

gures A1.1 through A1.4 show the logic of one Processor Board in detail.

re follows a short description of each of the Figures.

gure A1.1. I/O address decoding: The purpose of this circuitry is to ovide read and write control signals to the I/O data registers. The Host I/O processor, which views the I/O data registers as a part of its mory, provides an address and control signals for read and write. The ntrol Unit may issue a command, IOWRALL, to write the contents of the) data bus into all I/O data registers simultaneously.

gure A1.2. Control signal buffering, data buffering: This part contains ffers for control signals from the Control Unit, for data output from the mory Modules and for I/O data. All buffers are unidirectional, except the) data buffer, which is bidirectional.

gure A1.3. One Processing Element and Memory Module: Each board ntains eight of these. The arithmetic/logic unit is implemented as a PROM 1024 4-bit words. It gets its data inputs from the three flip-flops T, R, d C, from the Common Register of the Control Unit, and from a ltiplexer choosing one of eight sources. The outputs are stored in four p-flops, T, R, C, and X. The contents of R can be stored in one position the 4096-bit RWM.

formation can be shifted into or out from the I/O data register (U3i) om/to the memory. The I/O register is accessible from the outside by

means of an 8-bit data bus and read- and write signals, Ri* and Wi (see Figure A1.1).

Each PE contains part of the SELECT-FIRST network, which is shown in total in Figure A1.4.

Figure A1.4. The network for SELECT FIRST, SOME, and READ SELECTED I/O REGISTER: On the command "SELECT FIRST" (SELF*) issued by the Control Unit, only one (the first) T-flip-flop is to remain set to one, all others are reset. This is done by the left part of the figure. The circuitry shown to the right has two purposes. The first is to tell the Control Unit whether some or none of the T-flip-flops are set. The "SOME out" from one board is chained to "SOME in" on the next board. The second purpose is to put the contents of only one I/O register on the I/O data bus upon receiving the command "I/O Read Selected". The I/O register to be output is the one that belongs to the PE where the T-flip-flop is true.

Figure A1.1 I/O address decoding

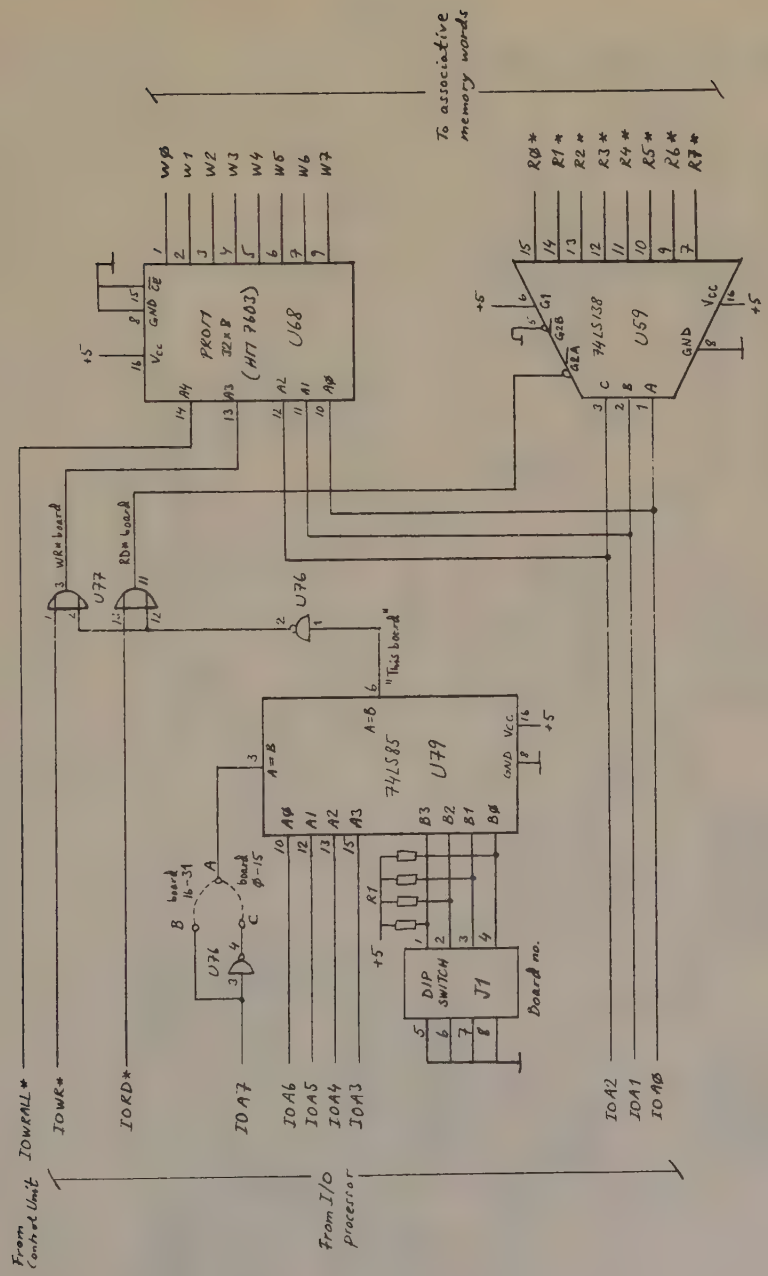


Figure A1.2 Control signal buffering, data buffering

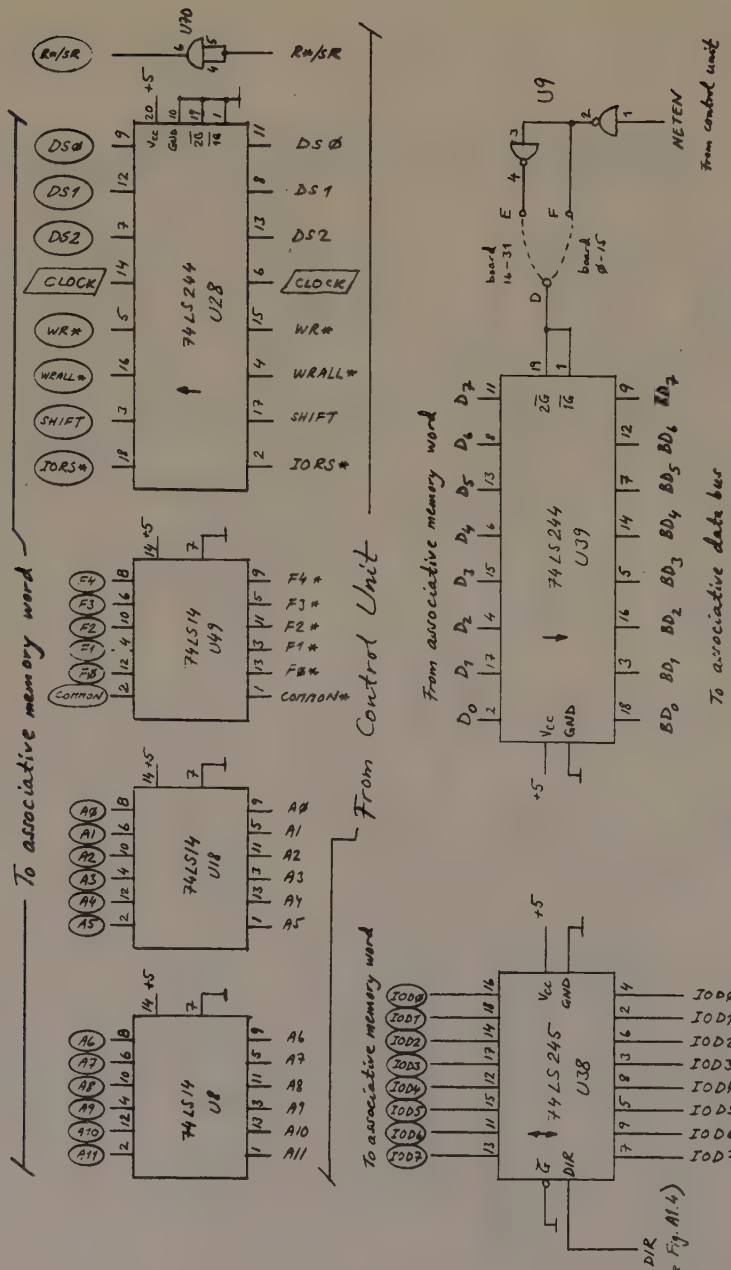


Figure A1.3 Processing Element and Memory Module no. i

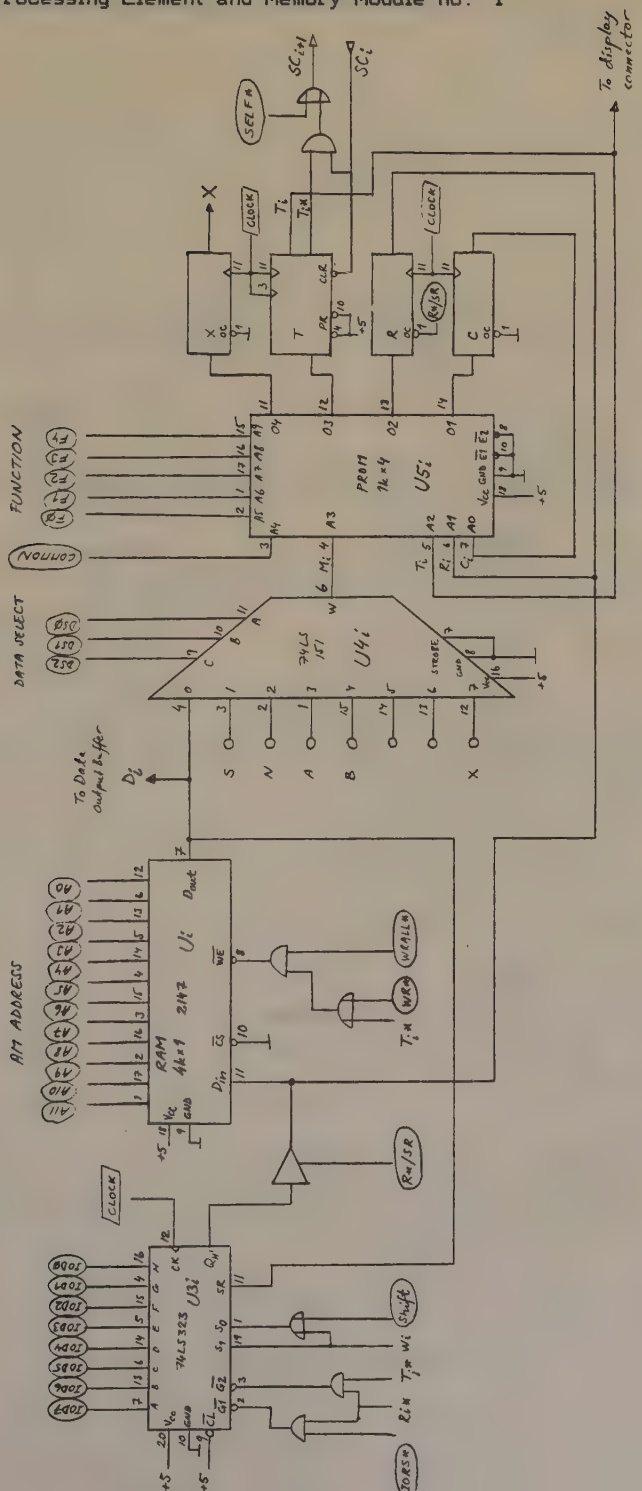
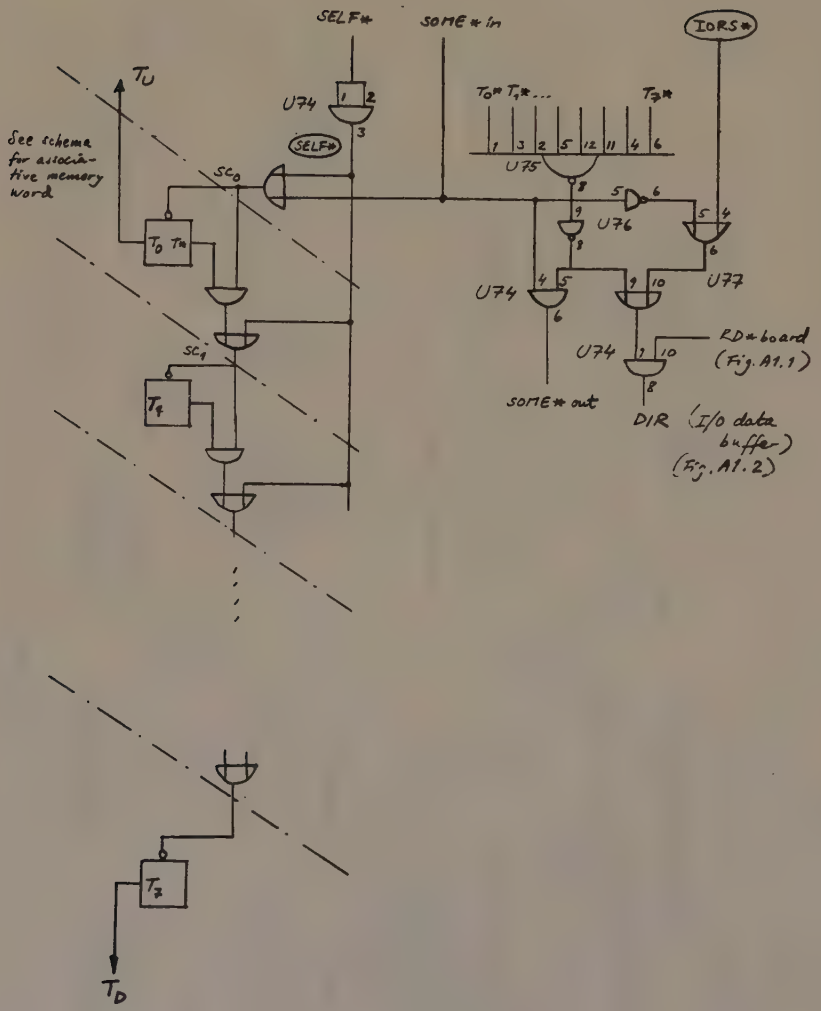


Figure A1.4 The network for SELECT FIRST, SOME, and READ SELECTED I/O REGISTER



A2. EXAMPLES OF MICROPROGRAMS

A2.1 In microprogramming assembly language

```
;*************************************************************
;     "Load Outdata All"
;
;     LDOA source
;
;     Outputs an 8-bit field to the I/O data registers
;*************************************************************

      LABEL   SET 07H               ;LOAD INTERNAL LOOP COUNTER
LDOA: DEF     LDCT,LDBD,RB1,PARAM1  ;LOAD BYTE-SLICE ADDRESS

      DEF     NEXT,RB1

      LABEL   SET LDOA1
      AM      SHIFT                 ;BIT-SLICE TO IO-REGISTER
LDOA1: DEF    INCB,RB1,RPCT         ;INCR ADDR POINTER, REPEAT
                                    ;...LOOP IF NOT FINISHED

      LABEL   SET OPFETCH
      DEF     CJP                   ;RETURN TO OPFETCH

;-------------------------------------------------------------


;*************************************************************
;     "Maximum of Field, Tagmasked"
;
;     MAXT field_address, length
;
;     Marks in the Tag flip-flops the word(s) (among the
;     selected ones) with largest value in the field
;*************************************************************

MAXT: DEF     LDBD,RB4,PARAM4       ;LOAD LENGTH
      DEF     ADAD,RA4,PARAM1,RB1   ;POINTER TO MSB+1 -> REG1

      DEF     DECB,RB1,NEXT,PUSH    ;SET UP LOOP
```

```
      AM       CMOT,D                ;SELECT ONES (OLD T TO X)
      DEF      RB1

      LABEL    SET MAXT1
      DEF      CJP,DECB,RB4,NONE     ;TEST IF SOME

      DEF      DECB,RB1,LOOP,ZAP     ;IF SO, GO TO NEXT BIT

      LABEL    SET OPFETCH
      DEF      CJP                   ;RET TO OPFETCH IF FINISHED

      AM       LTMA,X                ;IF NONE, RESTORE TAGS
MAXT1: DEF     DECB,RB1,LOOP,ZAP     ;.. AND GO TO NEXT BIT

      LABEL    SET OPFETCH
      DEF      CJP                   ;RET TO OPFETCH IF FINISHED

;-------------------------------------------------------------


;*************************************************************
;     "Compare Field to Common, Tagmasked"
;
;     COCOT field_address, length
;
;     Marks in the Tag flip-flops those words (of the
;     selected ones) where the contents of the examined
;     field is identical to the contents of the same
;     field in the Common Register
;*************************************************************

COCOT: DEF     LDLC,PARAM4
      DEF      LDBD,RB1,PARAM1

      DEF      DECLC,NEXT,RB1,PUSH   ;SET UP LOOP

      AM       CMCT,D                ;COMPARE MEM WITH COMMON
      DEF  DECLC,INCB,RB1,LOOP,ZLC   ;STEP POINTERS
                                     ;REPEAT UNTIL FINISHED
      LABEL    SET OPFETCH
      DEF      CJP                   ;THEN RETURN TO OPFETCH

;-------------------------------------------------------------
```

```
;*************************************************************
;     "Add Fields, Tagmasked"
;
;     ADDFT source1, source2, destination, length
;
;     Adds the contents one source field to another,
;     writing the result in the destination field
;*************************************************************

ADDFT: DEF    LDLC,PARAM4           ;LENGTH
       DEF    LDBD,RB3,PARAM3       ;DESTINATION
       DEF    LDBD,RB2,PARAM2       ;SOURCE 2
       DEF    LDBD,RB1,PARAM1       ;SOURCE 1

       DEF    DECB,RB3
       DEF    DECB,RB2

       AM     CCA                   ;CLEAR CARRY
       DEF    DECLC,NEXT,RB1,PUSH   ;NEXT, SET UP LOOP

       AM     LRMA,D                ;FETCH BIT FROM SOURCE 1
       DEF    INCB,RB2

       AM     ADMA,D                ;ADD BIT FROM SOURCE 2
       DEF    INCB,RB3

       AM     WRRT                  ;WRITE BIT IN DESTINATION
       DEF    DECLC,INCB,RB1,LOOP,ZLC ;LOOP IF NOT FINISHED

       LABEL  SET OPFETCH           ;ELSE RETURN TO OPFETCH
       DEF    CJP

;-------------------------------------------------------------
```

```
;*************************************************************
;     "Salt and Pepper Noise Removal"
;
;     SALTPEP source_image, dest_image, counter_address
;
;     Changes the value of a binary pixel if it differs from
;     seven or more of its eight neighbours. Two bit slices
;     are used to count to 2. The number of pixel columns is
;     assumed to be given in reg. D of the address processor
;*************************************************************

SALTPEP: DEF  LDBD,RB1,PARAM1      ;R1=BINARY SOURCE IMAGE ADDR.
     DEF      LDBD,RB3,PARAM3      ;R3=BINARY DEST IMAGE ADDR.
     DEF      LDBD,RB4,PARAM4      ;R4=COUNTER ADDR. (SCRATCH PAD)

     DEF      LDBA,RB2,RAD,NEXT    ;R2=NO OF PIXEL COLUMNS

     AM       CRA
     DEF      DECB,RB3

  ;SET UP LOOP AND INITIATE COUNTER FIELD
  ;=====================================
                                   ;CNT FIELD IS INIT TO 00.
                                   ;1ST POSITION IS MARKER FIELD
                                   ;R5=BITSLICE IN CNT FIELD

     DEF      LDBA,RB5,RA4,PUSH

     AM       WRRA
     DEF      INCB,RB5

     AM       WRRA
     DEF      RB1

  ;INCREMENT COUNTER ONE STEP FOR EACH MATCHING NEIGHBOUR

     AM       LRMA,D               ;FETCH PIXEL VALUE
     DEF      DECB,RB1

     AM       CORA                 ;INVERT IT
     DEF      RB1

     AM       XORRMA,A,NETEN       ;XOR WITH NORTHWEST
     DEF                           ;NEIGHBOUR. OLD R TO X

     LABEL    SET COUNT
     AM       LTMA,X               ;SAVE INVERTED PIXEL VALUE IN T
     DEF      CJS                  ;INCR COUNTER WHERE PIXEL
                                   ;EQUAL TO NEIGHBOUR
```

```
AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     RB1

LABEL   SET COUNT
AM      XORRMA,D            ;XOR WITH EAST NEIGHBOUR
DEF     CJS                 ;DECR COUNTER IF NON MATCH

AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     RB1

LABEL   SET COUNT
AM      XORRMA,B,NETEN      ;XOR WITH SOUTHEAST NEIGHBOUR
DEF     CJS                 ;DECR COUNTER IF NON MATCH

AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     INCB,RB1

DEF     RB1
LABEL   SET COUNT
AM      XORRMA,A,NETEN      ;XOR WITH NORTH NEIGHBOUR
DEF     CJS                 ;DECR COUNTER IF NON MATCH

AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     RB1

LABEL   SET COUNT
AM      XORRMA,B,NETEN      ;XOR WITH SOUTH NEIGHBOUR
DEF     CJS                 ;DECR COUNTER IF NON MATCH

AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     INCB,RB1

LABEL   SET COUNT
AM      XORRMA,A,NETEN      ;XOR WITH NORTHWEST NEIGHBOUR
DEF     CJS                 ;DECR COUNTER IF NON MATCH

AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     RB1

LABEL   SET COUNT
AM      XORRMA,D            ;XOR WITH WEST NEIGHBOUR
DEF     CJS                 ;DECR COUNTER IF NON MATCH

AM      LRTA                ;FETCH INVERTED PIXEL AGAIN
DEF     RB1

LABEL   SET COUNT
AM      XORRMA,B,NETEN      ;XOR WITH SOUTHWEST NEIGHBOUR
```

```
      DEF     CJS                 ;DECR COUNTER IF NON MATCH

  ;NOW, INVERT PIXEL VALUES WHERE MARKER FIELD =1

      AM      LRMA,D              ;GET PIXEL VALUE
      DEF     RB5

      AM      XORRMA,D            ;XOR WITH MARKER FIELD
      DEF     INCB,RB3

      AM      WRRA                ;WRITE IN DEST IMAGE
      DEF     DECB,RB2

      AM      CRA
      DEF     LDBA,RB5,RA4,LOOP,ZAP ;LOOP IF MORE PIXEL COLUMNS

  ;END OF MICROPROGRAM SALTPEP

      LABEL   SET OPFETCH
      DEF     CJP

  ;SUBROUTINE: ADD R TO COUNTER FIELD.

      AM      CCA                 ;CLEAR CARRY
COUNT: DEF    LDBA,RB5,RA4        ;LOAD BIT SLICE POINTER

      AM      ADMA,D              ;ADD R TO FIRST BIT.
      DEF     RB5

      AM      WRRA,CRA            ;WRITE RESULT BIT. CLEAR R
      DEF     INCB,RB5

      AM      ADMA,D              ;SECOND BIT
      DEF     RB5

      AM      ORRMA,D             ;OR WITH OLD SECOND BIT
      DEF     RB5

      AM      WRRA,CCA
      DEF     CRTN,RB1            ;RET (WITH CY=0)(AND RB1 OUT)

;-------------------------------------------------------------
```

```
;************************************************************
;     "Border finding"
;
;     BORDER8 source_image, dest_image, scratchpad_slice
;
;     Border points in a binary image, i.e. '1's with a '0' as
;     a neighbour, are marked. 8-adjacency is used to define
;     neighbour. The number of pixel columns is assumed to be
;     given in register D of the address processor
;************************************************************

BORDER8: DEF  LDBD,RB3,PARAM3       ;R3=BINARY DESTINATION IMAGE
      DEF     LDBD,RB4,PARAM4       ;R4=SCRATCH PAD BIT SLICE
      DEF     LDBA,RB5,RAD,         ;R5=NO OF PIXEL COLUMNS

      DEF     DECB,RB3

      DEF     LDBD,RB1,PARAM1       ;R1=BINARY SOURCE IMAGE

      AM      LTMA,D                ;READ BIT-SLICE
      DEF     RB1,PUSH,NEXT

      LABEL   SET BORD81
      AM      LRTA
      DEF     DECB,RB1,CJP,NONE     ;IF SOME '1' FORM AND-PRODUCT
                                    ;OF 9-PIXEL NEIGHBOURHOOD
      DATA    SET 02H
      AM      ANDRMA,D              ;AND WITH RIGHT NEIGHBOURS
      DEF     ADAD,RB1,RA1,PPL

      AM      ANDRMA,D              ;AND WITH LEFT NEIGHBOURS
      DEF     RB4

      AM      WRRA                  ;SAVE IN SCR PAD FOR VERT. AND
      DEF     RB4

      AM      ANDRMA,A,NETEN        ;AND WITH HORIZ. PRODUCT ABOVE
      DEF     RB4

      AM      ANDRMA,B,NETEN        ;  -   "   -            BELOW
      DEF     DECB,RB1

      AM      CORA                  ;NOW,'1' IN R INDICATE A '0' IN
      DEF     RB1                   ;THE NEIGHBOURHOOD

      AM      ANDRMA,D              ;IF THE PIX VALUE ALSO IS '1'..
      DEF     INCB,RB1

BORD81:       DEF     INCB,RB3
```

```
      AM       WRRA                  ;...THEN WRITE '1', ELSE '0'
      DEF      DECB,RB5

      DEF      RB1,LOOP,ZAP          ;LOOP IF MORE PIXEL COLUMNS

      LABEL    SET OPFETCH           ;ELSE RETURN TO OPFETCH
      DEF      CJP

;-------------------------------------------------------------
```

A2.2 In high-level microprogramming language

```
{*************************************************************}
{     "Multiply Fields, All"                                  }
{                                                             }
{     MULFA source1, source2, dest, length                    }
{                                                             }
{     Twos-complement multiplication                          }
{*************************************************************}

Module MULFAmic;
var s1,s2,d,k,b;

Microprogram MULFA(source1,source2,destination,length);
begin
   ltma(source2,direct); s1:=source1; d:=destination;
   iterate length times
   begin
      lrma(s1,direct); s1:=s1+1; andtra; wrra(d); d:=d+1;
   end;
   wrra(d); d:=d+1;
   cra;

   iterate length-2 times
   begin
      wrra(d); d:=d+1;
   end;

   k:=length-2; b:=length+1; d:=destination+1;
   iterate k times
   begin
      ltma(s2,direct); s2:=s2+1;
```

```
      call ADDFT(d,s1,d,b); d:=d+1;
   end;

   ltma(s2,direct);
   call SUBFT(d,s1,d,b);
end;

endmod.


{*****************************************************************}
{     "Convert to Canonical Signed Digit Form"                    }
{                                                                 }
{     CONVCSD source, destination, length                         }
{                                                                 }
{     Converts a binary coded integer to canonical                }
{     signed digit form                                           }
{*****************************************************************}

Module CONVCSDmic;

Microprogram CONVCSD(source, destination, length);
begin
   cca; cra;                {initiate state to zero}
   iterate length times
   begin
      xorrma(source,direct);
      wrra(destination); destination:=destination+1;
      lrma(source,direct); source:=source+1;
      adma(source,direct); lrca;
      wrra(destination); destination:=destination+1;
   end;
end;

endmod.
```

```
{*************************************************************}
{     "Outer Perimeter of Objects"                            }
{                                                             }
{     OUTPER (source, marker, dest, width)                    }
{                                                             }
{     Propagates bits in the marker field over points         }
{     where source field is zero. Finally, the result         }
{     is used to mark the outer perimeters of the objects     }
{*************************************************************}

Module OUTPERmic;

const ss=0FE0;                              {scan status}
var w,s,m,d;

Microprogram OUTPER(sim,mim,dim,iw);    {source image, marker image,..}
                                        {... dest image, image width}
begin
   while true do
   begin
      cra; wrra(ss);                        {clear scanstatus}
      w:=iw; s:=sim; m:=mim; sett;

      while SOME do
      begin                             {spread in rightmost column}
         lrma(m,above); orrma(m,below); ltra;
         andtmia(s,direct); andtmia(m,direct); lrta; wrrt(m);
      end;
      w:=w-1; s:=s+1;

      while w<>0 do                     {*** leftscan ***}
      begin
         lrma(m,direct); orrma(m,above); orrma(m,below); m:=m+1;
         orrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
         andtmia(m,direct); lrta; wrrt(m); orrma(ss,direct); wrra(ss);

         while SOME do
         begin
            lrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
            andtmia(m,direct); lrta; wrrt(m); orrma(ss,direct); wrra(ss);
         end;

         w:=w-1; s:=s+1;
      end;  {while w<>0 do}

      ltma(ss,direct);
      cca; if NONE then exit;           {if no change in the whole leftscan}
                                        {then finished spreading}
```

```
      cra; wrra(ss);                    {clear scanstatus}
      w:=iw; s:=s-1; sett;

      while SOME do                     {spread in leftmost column}
      begin
         lrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
         andtmia(m,direct); lrta; wrrt(m);
      end;
      w:=w-1; s:=s-1;

      while w<>0 do                     {*** rightscan ***}
      begin
         lrma(m,direct); orrma(m,above); orrma(m,below); m:=m-1;
         orrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
         andtmia(m,direct); lrta; wrrt(m); orrma(ss,direct); wrra(ss);

         while SOME do
         begin
            lrma(m,above); orrma(m,below); ltra; andtmia(s,direct);
            andtmia(m,direct); lrta; wrrt(m); orrma(ss,direct); wrra(ss);
         end;

         w:=w-1; s:=s-1;
      end;  {while w<>0 do}

      ltma(ss,direct);
      cca; if NONE then exit;           {if no change in the whole rightscan}
                                        {then finish spreading}
   end; {while true}

   w:=iw; s:=sim; m:=mim; d:=dim;

   while w<>0 do                        {Put border in dest image}
   begin
      ltma(s,direct); lrta;
      if SOME then
      begin
         andtmia(m,above); andtmia(m,below); m:=m-1;
         andtmia(m,direct); m:=m+2; andtmia(m,direct); cot;
         wrrt(d);
      end
      else m:=m+1;
      w:=w-1; s:=s+1; d:=d+1;
   end;
end;        {microprogram OUTPER}

endmod.
```

Bertil Svensson
LUCAS PROCESSOR ARRAY – Design and Applications

The ability to process large amounts of data very fast is generally considered as the most salient feature of modern digital computers. In the processing unit, data is usually dealt with only one item at a time. To increase speed further, new processor organizations that utilize parallelism are required.

The subject of this thesis is the design of a highly parallel computer. A prototype system, LUCAS (Lund University Content Addressable System), has been implemented and is described in detail. It comprises 128 bit-serial processors working in SIMD (Single Instruction stream, Multiple Data streams) mode. The processors are arranged as a linear array with perfect shuffle/exchange interconnection and nearest neighbour communication.

The thesis reports on application studies made. Algorithms for fast matrix multiplication are presented, as well as the detailed execution of the Fast Fourier Transform. It is shown how graph theoretic problems are solved efficiently on this kind of computer structure. Finally, the applicability of the machine in the area of image processing is demonstrated through the implementation of several image operations. The results obtained are applicable on other machines with similar structure.

Two related theses report further on design issues, programming and application studies:

Christer Fernström: The LUCAS Associative Array Processor and its Programming Environment, PhD thesis 1983.

Ivan Kruzela: An Associative Array Processor Supporting a Relational Algebra, PhD thesis 1983.